Data Skills for Reproducible Research
v2

2021-09-25

Contents

Overview

This book provides an overview of skills needed for reproducible research and open science using the statistical programming language R and tidyverse packages. It covers data visualisation, data tidying and wrangling, archiving, iteration and functions, probability and data simulations, general linear models, and reproducible workflows.

The book is also available in PDF, ePub and Kindle formats, but these may have odd formatting where the onlne book has interactive elements like quizzes and shiny apps.

Resources

- Data Skills Videos Each chapter has several short video lectures for the main learning outcomes. The videos are captioned and watching with the captioning on is a useful way to learn the jargon of computational reproducibility. If you cannot access YouTube, the videos are available by request.

- reprores This is a custom R package for this course. You can install it with the code below. It will download all of the packages that are used in the book, along with an offline copy of this book, the shiny apps used in the book, and the exercises.

  ```
  devtools::install_github("psyteachr/reprores-v2")
  ```

- glossary Coding and statistics both have a lot of specialist terms. Throughout this book, jargon will be linked to the glossary.

How to learn data skills

Learning data skills is kind of like having a gym membership (HT to Phil McAleer for the analogy). You'll be given state-of-the-art equipment to use and instructions for how to use them, but your data skills won't get any stronger unless you practice.

Data skills do not require you to memorise lots of code. You will be introduced to many different functions, but the main skill to learn is how to efficiently find the information you need. This will require getting used to the structure of help files and cheat sheets, learning how to Goggle your problem and choose a helpful solution, and learning how to read error messages.

Learning to code involves making a **lot** of mistakes. These mistakes are completely essential to the process, so try not to feel too frustrated. Many of the chapter exercises will give you broken code to fix so you get experience seeing what common errors look like. As you become a more experienced coder, you might not make fewer errors, but you'll recover from them much faster.

I found a bug!

This book is a work in progress, so you might find errors. Please help me fix them! The best way is to open an issue on github that describes the error, but you can also email Lisa.

Other Resources

- RStudio Cheat Sheets
- Improving Pedagogy through Registered Reports
- Learning Statistics with R by Navarro
- R for Data Science by Grolemund and Wickham
- Improving your statistical inferences on Coursera
- swirl
- R for Reproducible Scientific Analysis
- codeschool.com
- datacamp
- Style guide for R programming
- #rstats on twitter highly recommended!

Chapter 1

Getting Started

1.1 Learning Objectives

1. Understand the components of the RStudio IDE (video)
2. Type commands into the console (video)
3. Understand coding terms and function syntax (video)
4. Install a package (video)
5. Know the methods for getting help

1.2 R and RStudio

R is a programming environment for data processing and statistical analysis. We use R in Psychology at the University of Glasgow to promote reproducible research. This refers to being able to document and reproduce all of the steps between raw data and results. R allows you to write scripts that combine data files, clean data, and run analyses. There are many other ways to do this, including writing SPSS syntax files, but we find R to be a useful tool that is free, open source, and commonly used by research psychologists.

> See Appendix **??** for more information on on how to install R and associated programs.

1.2.1 The Base R Console

If you open up the application called R, you will see an "R Console" window that looks something like this.

You can close R and never open it again. We'll be working entirely in RStudio in this class.

> ALWAYS REMEMBER: Launch R though the RStudio IDE
> Launch (RStudio.app), not (R.app).

1.2.2 RStudio

RStudio is an Integrated Development Environment (IDE). This is a program that serves as a text editor, file manager, and provides many functions to help you read and write R code.
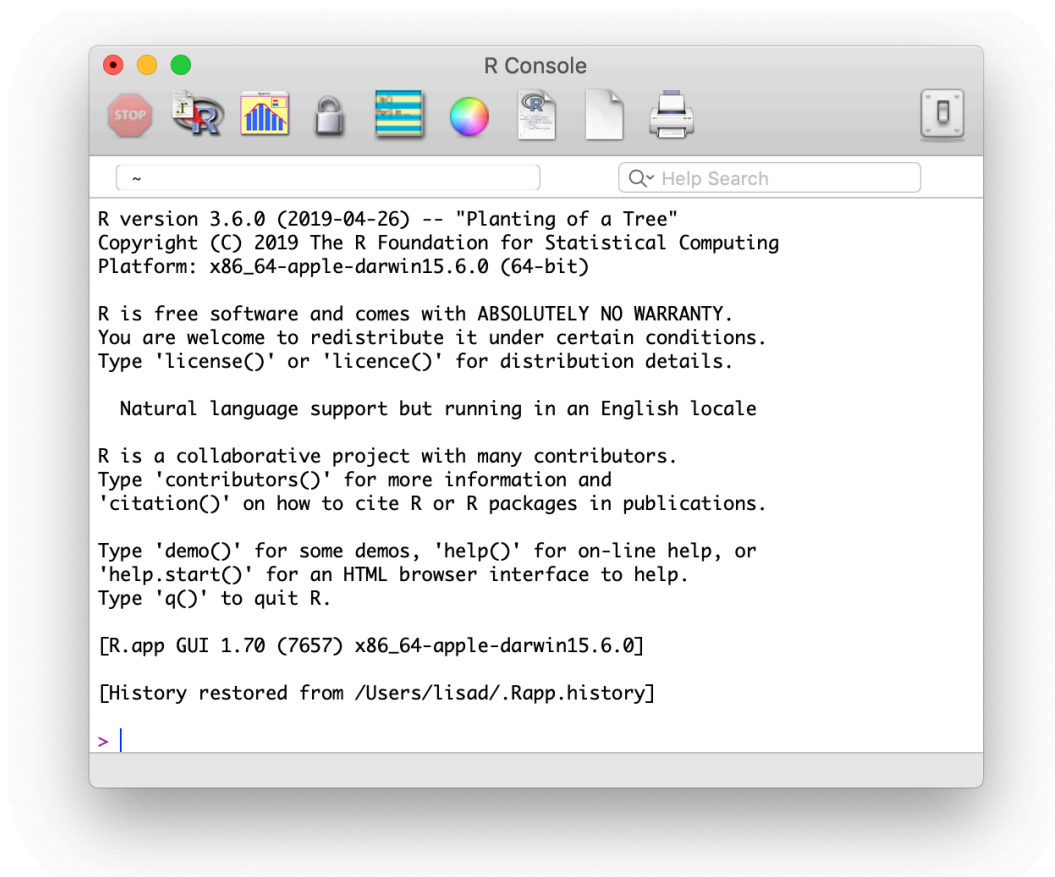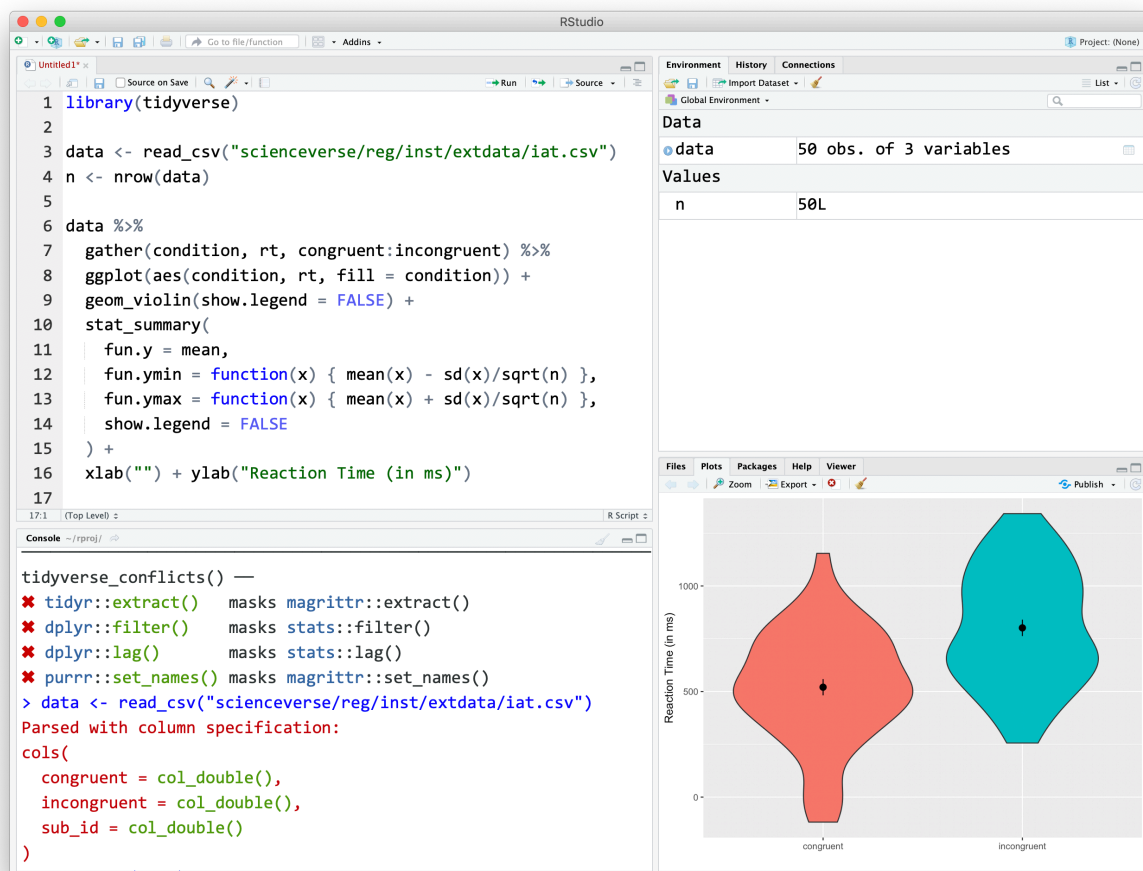
Figure 1.1: The R Console window.

Figure 1.2: The RStudio IDE

RStudio is arranged with four window panes. By default, the upper left pane is the **source pane**, where you view and edit source code from files. The bottom left pane is usually the **console pane**, where you can type in commands and view output messages. The right panes have several different tabs that show you information about your code. You can change the location of panes and what tabs are shown under `Preferences > Pane Layout`.

1.2.3   Configure RStudio

In this class, you will be learning how to do reproducible research. This involves writing scripts that completely and transparently perform some analysis from start to finish in a way that yields the same result for different people using the same software on different computers. Transparency is a key value of science, as embodied in the "trust but verify" motto.

When you do things reproducibly, others can understand and check your work. This benefits science, but there is a selfish reason, too: the most important person who will benefit from a reproducible script is your future self. When you return to an analysis after two weeks of vacation, you will thank your earlier self for doing things in a transparent, reproducible way, as you can easily pick up right where you left off.

There are two tweaks that you should do to your RStudio installation to maximize reproducibility. Go to `Global Options...` under the `Tools` menu (Cmd-,), and uncheck the box that says `Restore .RData into workspace at startup`. If you keep things around in your workspace, things will get messy, and unexpected things will happen. You should always start with a clear workspace. This also means that you never want to save your workspace when you exit, so set this to `Never`. The only thing you want to save are your scripts.
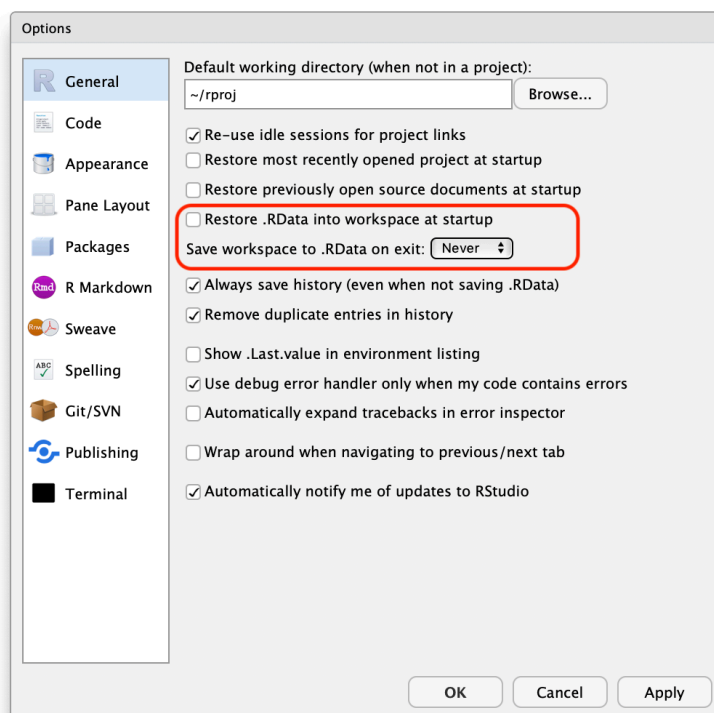

Figure 1.3: Alter these settings for increased reproducibility.

> Your settings should have:
>
> - Restore .RData into workspace at startup: CheckedNot Checked
> - Save workspace to .RData on exit: AlwaysNeverAsk

1.3 Console commands

We are first going to learn about how to interact with the console. In general, you will be developing R scripts or R Markdown files, rather than working directly in the console window. However, you can consider the console a kind of "sandbox" where you can try out lines of code and adapt them until you get them to do what you want. Then you can copy them back into the script editor.

Mostly, however, you will be typing into the script editor window (either into an R script or an R Markdown file) and then sending the commands to the console by placing the cursor on the line and holding down the Ctrl key while you press Enter. The Ctrl+Enter key sequence sends the command in the script to the console.

One simple way to learn about the R console is to use it as a calculator. Enter the lines of code below and see if your results match. Be prepared to make lots of typos (at first).

```r
1 + 1
```

```
## [1] 2
```

The R console remembers a history of the commands you typed in the past. Use the up and down arrow keys on your keyboard to scroll backwards and forwards through your history. It's a lot faster than re-typing.

```r
1 + 1 + 3
```

```
## [1] 5
```

You can break up mathematical expressions over multiple lines; R waits for a complete expression before processing it.

```r
## here comes a long expression
## let's break it over multiple lines
1 + 2 + 3 + 4 + 5 + 6 +
    7 + 8 + 9 +
    10
```

```
## [1] 55
```

Text inside quotes is called a string.

```r
"Good afternoon"
```

```
## [1] "Good afternoon"
```

You can break up text over multiple lines; R waits for a close quote before processing it. If you want to include a double quote inside this quoted string, escape it with a backslash.

```
africa <- "I hear the drums echoing tonight
But she hears only whispers of some quiet conversation
She's coming in, 12:30 flight
The moonlit wings reflect the stars that guide me towards salvation
I stopped an old man along the way
Hoping to find some old forgotten words or ancient melodies
He turned to me as if to say, \"Hurry boy, it's waiting there for you\"

- Toto"

cat(africa) # cat() prints the string
```

```
## I hear the drums echoing tonight
## But she hears only whispers of some quiet conversation
## She's coming in, 12:30 flight
## The moonlit wings reflect the stars that guide me towards salvation
## I stopped an old man along the way
## Hoping to find some old forgotten words or ancient melodies
## He turned to me as if to say, "Hurry boy, it's waiting there for you"
##
## - Toto
```

1.4 Coding Terms

1.4.1 Objects

Often you want to store the result of some computation for later use. You can store it in an object (also sometimes called a variable). An object in R:

- contains only letters, numbers, full stops, and underscores
- starts with a letter or a full stop and a letter
- distinguishes uppercase and lowercase letters (`rickastley` is not the same as `RickAstley`)

The following are valid and different objects:

- `songdata`
- `SongData`
- `song_data`
- `song.data`
- `.song.data`
- `never_gonna_give_you_up_never_gonna_let_you_down`

The following are not valid objects:

- `_song_data`
- `1song`
- `.1song`
- `song data`
- `song-data`

Use the assignment operator<-' to assign the value on the right to the object named on the left.

```r
## use the assignment operator '<-'
## R stores the number in the object
x <- 5
```

Now that we have set x to a value, we can do something with it:

```r
x * 2

## R evaluates the expression and stores the result in the object boring_calculation
boring_calculation <- 2 + 2
```

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

Note that it doesn't print the result back at you when it's stored. To view the result, just type the object name on a blank line.

```r
boring_calculation
```

```
## [1] 4
```

Once an object is assigned a value, its value doesn't change unless you reassign the object, even if the objects you used to calculate it change. Predict what the code below does and test yourself:

```r
this_year <- 2019
my_birth_year <- 1976
my_age <- this_year - my_birth_year
this_year <- 2020
```

> After all the code above is run:
>
> - `this_year` = 4344197620192020
> - `my_birth_year` = 4344197620192020
> - `my_age` = 4344197620192020

1.4.2 The environment

Any time you assign something to a new object, R creates a new entry in the global environment. Objects in the global environment exist until you end your session; then they disappear forever (unless you save them).

Look at the **Environment** tab in the upper right pane. It lists all of the objects you have created. Click the broom icon to clear all of the objects and start fresh. You can also use the following functions in the console to view all objects, remove one object, or remove all objects.

```r
ls()             # print the objects in the global environment
rm("x")          # remove the object named x from the global environment
rm(list = ls())  # clear out the global environment
```

> In the upper right corner of the Environment tab, change **List** to **Grid**. Now you can see the type, length, and size of your objects, and reorder the list by any of these attributes.

1.4.3 Whitespace

R mostly ignores whitespace: spaces, tabs, and line breaks. This means that you can use whitespace to help you organise your code.

```r
# a and b are identical
a <- list(ctl = "Control Condition", exp1 = "Experimental Condition 1", exp2 = "Experimental Condition

# but b is much easier to read
b <- list(ctl  = "Control Condition",
          exp1 = "Experimental Condition 1",
          exp2 = "Experimental Condition 2")
```

When you see `>` at the beginning of a line, that means R is waiting for you to start a new command. However, if you see a `+` instead of `>` at the start of the line, that means R is waiting for you to finish a command you started on a previous line. If you want to cancel whatever command you started, just press the Esc key in the console window and you'll get back to the `>` command prompt.

```r
# R waits until next line for evaluation
(3 + 2) *
    5
```

```
## [1] 25
```

It is often useful to break up long functions onto several lines.

```r
cat("3, 6, 9, the goose drank wine",
    "The monkey chewed tobacco on the streetcar line",
    "The line broke, the monkey got choked",
    "And they all went to heaven in a little rowboat",
    sep = "  \n")
```

```
## 3, 6, 9, the goose drank wine
## The monkey chewed tobacco on the streetcar line
## The line broke, the monkey got choked
## And they all went to heaven in a little rowboat
```

1.4.4 Function syntax

A lot of what you do in R involves calling a function and storing the results. A function is a named section of code that can be reused.

For example, **sd** is a function that returns the standard deviation of the vector of numbers that you provide as the input argument. Functions are set up like this:

```r
function_name(argument1, argument2 = "value")
```

The arguments in parentheses can be named (e.g., `argument1 = 10`) or you can skip the names if you put them in the exact same order that they're defined in the function. You can check this by typing `?sd` (or whatever function name you're looking up) into the console and the Help pane will show you the default order under **Usage**. You can skip arguments that have a default value specified.

Most functions return a value, but may also produce side effects like printing to the console.

To illustrate, the function `rnorm()` generates random numbers from the standard normal distribution. The help page for `rnorm()` (accessed by typing `?rnorm` in the console) shows that it has the syntax

```r
rnorm(n, mean = 0, sd = 1)
```

where `n` is the number of randomly generated numbers you want, `mean` is the mean of the distribution, and `sd` is the standard deviation. The default mean is 0, and the default standard deviation is 1. There is no default for `n`, which means you'll get an error if you don't specify it:

```r
rnorm()
```

```
## Error in rnorm(): argument "n" is missing, with no default
```

If you want 10 random numbers from a normal distribution with mean of 0 and standard deviation, you can just use the defaults.

```r
rnorm(10)
```

```
##  [1]  0.3644909  1.0551302  2.0720850  0.6152214 -0.2526608 -0.4186245
##  [7] -0.1663370  0.1821596  1.1473016  0.9838531
```

If you want 10 numbers from a normal distribution with a mean of 100:

```r
rnorm(10, 100)
```

```
##  [1] 100.32126  99.03195 100.08633  98.23100  98.76142  99.97122 101.28922
##  [8] 100.83947  99.28170 100.09866
```

This would be an equivalent but less efficient way of calling the function:

```r
rnorm(n = 10, mean = 100)
```

```
##  [1] 100.13668 100.11537  99.73778 101.14736 101.39270 100.92577  99.07943
##  [8] 100.45524 100.72273 100.62765
```

We don't need to name the arguments because R will recognize that we intended to fill in the first and second arguments by their position in the function call. However, if we want to change the default for an argument coming later in the list, then we need to name it. For instance, if we wanted to keep the default `mean = 0` but change the standard deviation to 100, we would do it this way:

```r
rnorm(10, sd = 100)
```

```
##  [1] -145.718925 -246.111293   45.212757   -7.484518   63.115975  128.406898
##  [7]   35.714157  170.416315  -39.284274  -65.863168
```

Some functions give a list of options after an argument; this means the default value is the first option. The usage entry for the power.t.test() function looks like this:

```
power.t.test(n = NULL, delta = NULL, sd = 1, sig.level = 0.05,
             power = NULL,
             type = c("two.sample", "one.sample", "paired"),
             alternative = c("two.sided", "one.sided"),
             strict = FALSE, tol = .Machine$double.eps^0.25)
```

- What is the default value for `sd`? 1NULLtwo.sample0.05
- What is the default value for `type`? one.samplepairedNULLtwo.sample
- Which is equivalent to power.t.test(100, 0.5)?
  power.t.test() power.t.test(n = 100) power.t.test(100, 0.5, sig.level = 1, sd = 0.05)
  power.t.test(delta = 0.5, n = 100)

1.5 Add-on packages

One of the great things about R is that it is **user extensible**: anyone can create a new add-on software package that extends its functionality. There are currently thousands of add-on packages that R users have created to solve many different kinds of problems, or just simply to have fun. There are packages for data visualisation, machine learning, neuroimaging, eyetracking, web scraping, and playing games such as Sudoku.

Add-on packages are not distributed with base R, but have to be downloaded and installed from an archive, in the same way that you would, for instance, download and install PokemonGo on your smartphone.

The main repository where packages reside is called CRAN, the Comprehensive R Archive Network. A package has to pass strict tests devised by the R core team to be allowed to be part of the CRAN archive. You can install from the CRAN archive through R using the install.packages() function.

There is an important distinction between **installing** a package and **loading** a package.

1.5.1 Installing a package

This is done using install.packages(). This is like installing an app on your phone: you only have to do it once and the app will remain installed until you remove it. For instance, if you want to use PokemonGo on your phone, you install it once from the App Store or Play Store, and you don't have to re-install it each time you want to use it. Once you launch the app, it will run in the background until you close it or restart your phone. Likewise, when you install a package, the package will be available (but not *loaded*) every time you open up R.

> You may only be able to permanently install packages if you are using R on your own system; you may not be able to do this on public workstations if you lack the appropriate privileges.

Install the esquisse package on your system. This package lets you create plots interactively and copy the code needed to make them reproducibly.

```
# type this in the console pane
install.packages("esquisse")
```

If you don't already have packages like ggplot2 and shiny installed, it will also install these **dependencies** for you. If you don't get an error message at the end, the installation was successful.

Never install a package from inside a script. Only do this from the console pane.

1.5.2 Loading a package

This is done using library(packagename). This is like **launching** an app on your phone: the functionality is only there where the app is launched and remains there until you close the app or restart. Likewise, when you run library(packagename) within a session, the functionality of the package referred to by `packagename` will be made available for your R session. The next time you start R, you will need to run the library() function again if you want to access its functionality.

You can load the functions in esquisse for your current R session as follows:

```
library(esquisse)
```

You might get some red text when you load a package, this is normal. It is usually warning you that this package has functions that have the same name as other packages you've already loaded.

Now you can run the function esquisse::esquisser(), which runs an interactive plotting example on the built-in dataset `diamonds` from the ggplot2 package.

```
esquisse::esquisser(ggplot2::diamonds)
```

You can use the convention `package::function()` to indicate in which add-on package a function resides. For instance, if you see readr::read_csv(), that refers to the function read_csv() in the readr add-on package.
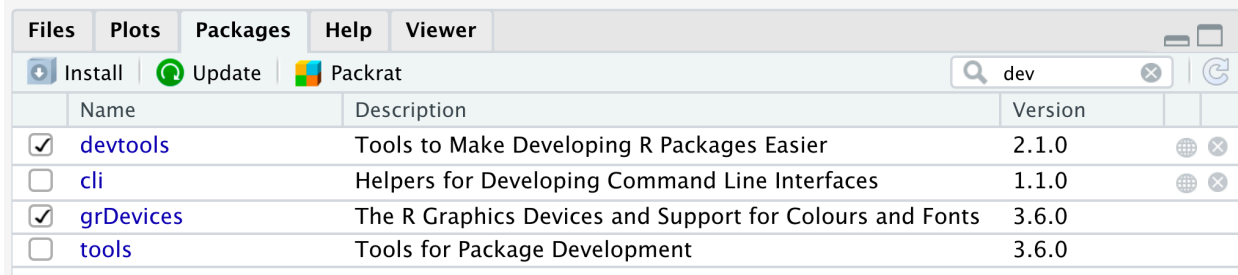
1.5.3 Tidyverse

tidyverseis a meta-package that loads several packages we'll be using in almost every script:

- ggplot2, for data visualisation (Chapter **??**)
- readr, for data import (Chapter **??**)
- tibble, for tables (Chapter **??**)
- tidyr, for data tidying (Chapter **??**)
- dplyr, for data manipulation (Chapter **??**)
- purrr, for repeating things (Chapter **??**)
- stringr, for strings
- forcats, for factors

1.5.4 Install from GitHub

Many R packages are not yet on CRAN because they are still in development. Increasingly, datasets and code for papers are available as packages you can download from github. You'll need to install the devtools package to be able to install packages from github. Check if you have a package installed by trying to load it (e.g., if you don't have devtools installed, library(devtools) will display an error message) or by searching for it in the packages tab in the lower right pane. All listed packages are installed; all checked packages are currently loaded.

Figure 1.4: Check installed and loaded packages in the packages tab in the lower right pane.

```
# install devtools if you get
# Error in loadNamespace(name) : there is no package called devtools
# install.packages("devtools")
devtools::install_github("psyteachr/reprores-v2")
```

After you install the reprores package, load it using the library() function. You can then try out some of the functions below.

```
library(reprores)

# opens a local copy of this book in your web browser
book()

# opens a shiny app that lets you see how simulated data would look in different plot styles
app("plotdemo")

# creates and opens a file containing the exercises for this chapter
exercise(1)
```

> How many different ways can you find to discover what functions are available in the reprores package?

reprores contains datasets that we will be using in future lessons. getdata() creates a directory called data with all of the class datasets.

```
# loads the disgust dataset
data("disgust")

# shows the documentation for the built-in dataset `disgust`
?disgust

# saves datasets into a "data" folder in your working directory
getdata("data")
```

1.6   Getting help

You will feel like you need a *lot* of help when you're starting to learn. This won't really go away, and it isn't supposed to. Experienced coders are also constantly looking things up; it's impossible to memorise

everything. The goal is to learn enough about the structure of R that you can look things up quickly. This is why there is so much specialised jargon in coding; it's easier to google "adding vectors in R" than "adding lists of things that are the same kind of data in R".

1.6.1 Function Help

Start up help in a browser using the function `help.start()`.

If a function is in base R or a loaded package, you can use the `help("function_name")` function or the `?function_name` shortcut to access the help file. If the package isn't loaded, specify the package name as the second argument to the help function.

```r
# these methods are all equivalent ways of getting help
help("rnorm")
?rnorm
help("rnorm", package="stats")
```

When the package isn't loaded or you aren't sure what package the function is in, use the shortcut `??function_name`.

- What is the first argument to the `mean` function? trimna.rmmeanx
- What package is `read_excel` in? readrreadxlbasestats

1.6.2 Googling

If the function help doesn't help, or you're not even sure what function you need, try Googling your question. It will take some practice to be able to use the right jargon in your search terms to get what you want. It helps to put "R" or "rstats", or "tidyverse" in the search text, or the name of the relevant package, like ggplot2.

1.6.3 Vignettes

Many packages, especially tidyverse ones, have helpful websites with vignettes explaining how to use their functions. Some of the vignettes are also available inside R.

```r
# opens a list of available vignettes
vignette(package = "ggplot2")

# opens a specific vignette in the Help pane
vignette("ggplot2-specs", package = "ggplot2")
```

1.7 Glossary

Each chapter ends with a glossary table defining the jargon introduced in this chapter. The links below take you to the glossary book, which you can also download for offline use.

```r
# install the glossary package (only once)
devtools::install_github("psyteachr/glossary")

# open the glossary offline
glossary::book()
```

term	definition
argument	A variable that provides input to a function.
assignment-operator	The symbol <-, which functions like = and assigns the value on the right to the object on the left
base-r	The set of R functions that come with a basic installation of R, before you add external packages
console	The pane in RStudio where you can type in commands and view output messages.
cran	The Comprehensive R Archive Network: a network of ftp and web servers around the world that s
escape	Include special characters like " inside of a string by prefacing them with a backslash.
factor	A data type where a specific set of values are stored with labels; An explanatory variable manipula
function	A named section of code that can be reused.
global-environment	The interactive workspace where your script runs
ide	Integrated Development Environment: a program that serves as a text editor, file manager, and pr
normal-distribution	A symmetric distribution of data where values near the centre are most probable.
object	A word that identifies and stores the value of some data for later use.
package	A group of R functions.
panes	RStudio is arranged with four window "panes".
r-markdown	The R-specific version of markdown: a way to specify formatting, such as headers, paragraphs, list
reproducible-research	Research that documents all of the steps between raw data and results in a way that can be verifie
script	A plain-text file that contains commands in a coding language, such as R.
standard-deviation	A descriptive statistic that measures how spread out data are relative to the mean.
string	A piece of text inside of quotes.
variable	A word that identifies and stores the value of some data for later use.
vector	A type of data structure that collects values with the same data type, like T/F values, numbers, o
whitespace	Spaces, tabs and line breaks

1.8 Further Resources

- Chapter 1: Introduction in *R for Data Science*
- RStudio IDE Cheatsheet
- RStudio Cloud

Chapter 2

Reproducible Workflows

2.1 Learning Objectives

Basic

1. Organise a project (video)
2. Create and compile an Rmarkdown document (video)
3. Edit the YAML header to add table of contents and other options
4. Include a table
5. Include a figure
6. Report the output of an analysis using inline R
7. Add a bibliography and in-line citations

Intermediate

8. Output doc and PDF formats
9. Format tables using `kableExtra`

2.2 Setup

```
library(tidyverse)
library(knitr)
```

2.3 Why learn reproducible reports?

Have you ever worked on a report, creating a summary table for the demographics, making beautiful plots, getting the analysis just right, and copying all the relevant numbers into your manuscript, only to find out that you forgot to exclude a test run and have to redo everything?

An R Markdown document produces a reproducible report that fixes this problem. Although this requires a bit of extra effort at the start, it will more than pay you back by allowing you to update your entire report with the push of a button whenever anything changes.

Studies also show that many, if not most, papers in the scientific literature have reporting errors. For example, more than half of over 250,000 psychology papers published between 1985 and 2013 have at least one value that is statistically incompatible (e.g., a p-value that is not possible given a t-value and degrees of freedom) (**?**). Reproducible reports help avoid transcription and rounding errors.

We will make reproducible reports following the principles of literate programming. The basic idea is to have the text of the report together in a single document along with the code needed to perform all analyses and generate the tables. The report is then "compiled" from the original format into some other, more portable format, such as HTML or PDF. This is different from traditional cutting and pasting approaches where, for instance, you create a graph in Microsoft Excel or a statistics program like SPSS and then paste it into Microsoft Word.

2.4 Organising a project

First, we need to get organised. Projects in RStudio are a way to group all of the files you need for one project. Most projects include scripts, data files, and output files like the PDF report created by the script or images.

2.4.1 File System

Modern computers tend to hide the file system from users, but we need to understand a little bit about how files are stored on your computer in order to get a script to find your data. Your computer's file system is like a big box (or directory) that contains both files and smaller boxes, or "subdirectories". You can specify the location of a file with its name and the names of all the directories it is inside.

For example, if Lisa is looking for a file called `report.Rmd`on their Desktop, they can specify the full file path like this: `/Users/lisad/Desktop/report.Rmd`, because the `Desktop` directory is inside the `lisad` directory, which is inside the `Users` directory, which is located at the base of the whole file system. If that file was on *your* desktop, you would probably have a different path unless your user directory is also called `lisad`. You can also use the `~` shortcut to represent the user directory of the person who is currently logged in, like this: `~/Desktop/report.Rmd`.

2.4.2 Working Directory

Where should you put all of your files? You usually want to have all of your scripts and data files for a single project inside one folder on your computer, the working directory for that project. You can organise files in subdirectories inside this main project directory, such as putting all raw data files in a directory called data and saving any image files to a directory called images.

Your script should only reference files in three locations, using the appropriate format.

Where	Example
on the web	"https://psyteachr.github.io/ads-v1/data/widgets_gadgets.xlsx"
in the working directory	"widgets_gadgets.xlsx"
in a subdirectory	"data/widgets_gadgets.xlsx"

Never set or change your working directory in a script.

R Markdown files will automatically use the same directory the .Rmd file is in as the working directory.

If your script needs a file in a subdirectory of your working directory, such as, data/widgets_gadgets.xlsx, load it in using a relative path so that it is accessible if you move the working directory to another location or computer:

```
dat <- read_csv("data/widgets_gadgets.xlsx")  # correct
```

Do not load it in using an absolute path:

```
dat <- read_csv("C:/My Files/2020-2021/data/widgets_gadgets.xlsx")   # wrong
```

> Also note the convention of using forward slashes, unlike the Windows-specific convention of using backward slashes. This is to make references to files work for everyone, regardless of their operating system.

2.4.3 Naming Things

Name files so that both people and computers can easily find things. Here are some important principles:

- file and directory names should only contain letters, numbers, dashes, and underscores, with a full stop (.) between the file name and extension (that means no spaces!)
- be consistent with capitalisation (set a rule to make it easy to remember, like always use lowercase)
- use underscores (_) to separate parts of the file name, and dashes (-) to separate words in a section
- name files with a pattern that alphabetises in a sensible order and makes it easy for you to find the file you're looking for
- prefix a filename with an underscore to move it to the top of the list, or prefix all files with numbers to control their order

For example, these file names are a mess:

- report.doc
- report final.doc
- Data (Customers) 11-15.xls
- Customers Data Nov 12.xls
- final report2.doc
- project notes.txt
- Vendor Data November 15.xls

Here is one way to structure them so that similar files have the same structure and it's easy for a human to scan the list or to use code to find relevant files. See if you can figure out what the last one should be.

- _project-notes.txt
- report_v1.doc
- report_v2.doc
- report_v3.doc
- data_customer_2021-11-12.xls
- data_customer_2021-11-15.xls
- vendor-data_2021-11-15.xlsdata-vendor-2021_11_15.xlsdata_vendor_2021-11-15.xlsdata_2021-11-15_vendor.xls

> Think of other ways to name the files above. Look at some of your own project files and see what you can improve.

2.4.4   Start a Project

Now that we understand how the file system work and how to name things to make it easier for scripts to access them, we're ready to make our first project.

First, make a new directory where you will keep all of your materials for this class. You can set this directory to be the default working directory under the General tab of the Global Options. This means that files will be saved here by default if you aren't working in a project.

Next, choose `New Project...` under the `File` menu to create a new project called 02-reports. Make sure you save it inside the directory you just made. RStudio will restart itself and open with this new project directory as the working directory.

```
include_graphics(c("images/repro/new_proj_1.png",
                   "images/repro/new_proj_2.png",
                   "images/repro/new_proj_3.png"))
```
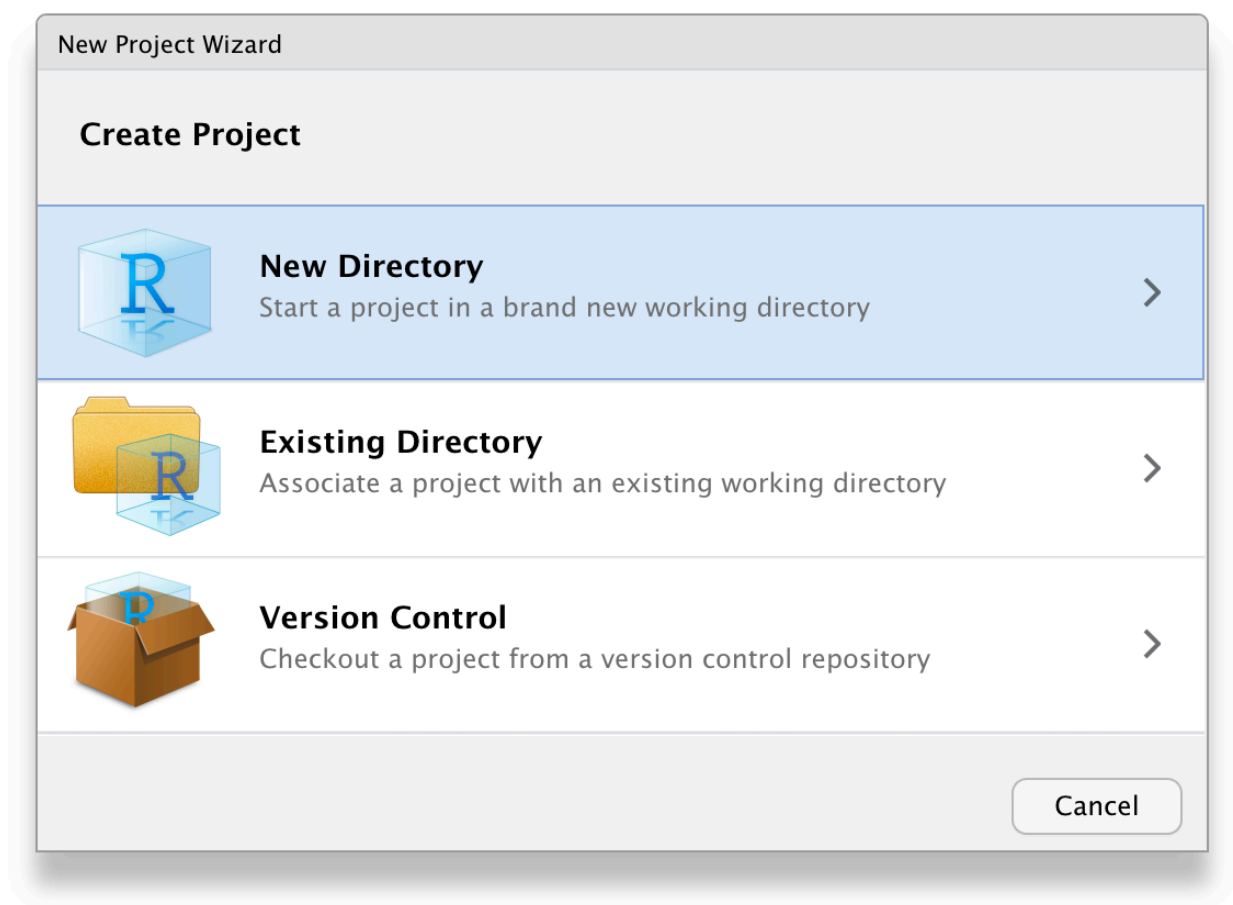

Figure 2.1: Starting a new project.

Click on the Files tab in the lower right pane to see the contents of the project directory. You will see a file called `02-reports.Rproj`, which is a file that contains all of the project information.You can double-click on it to open up the project.
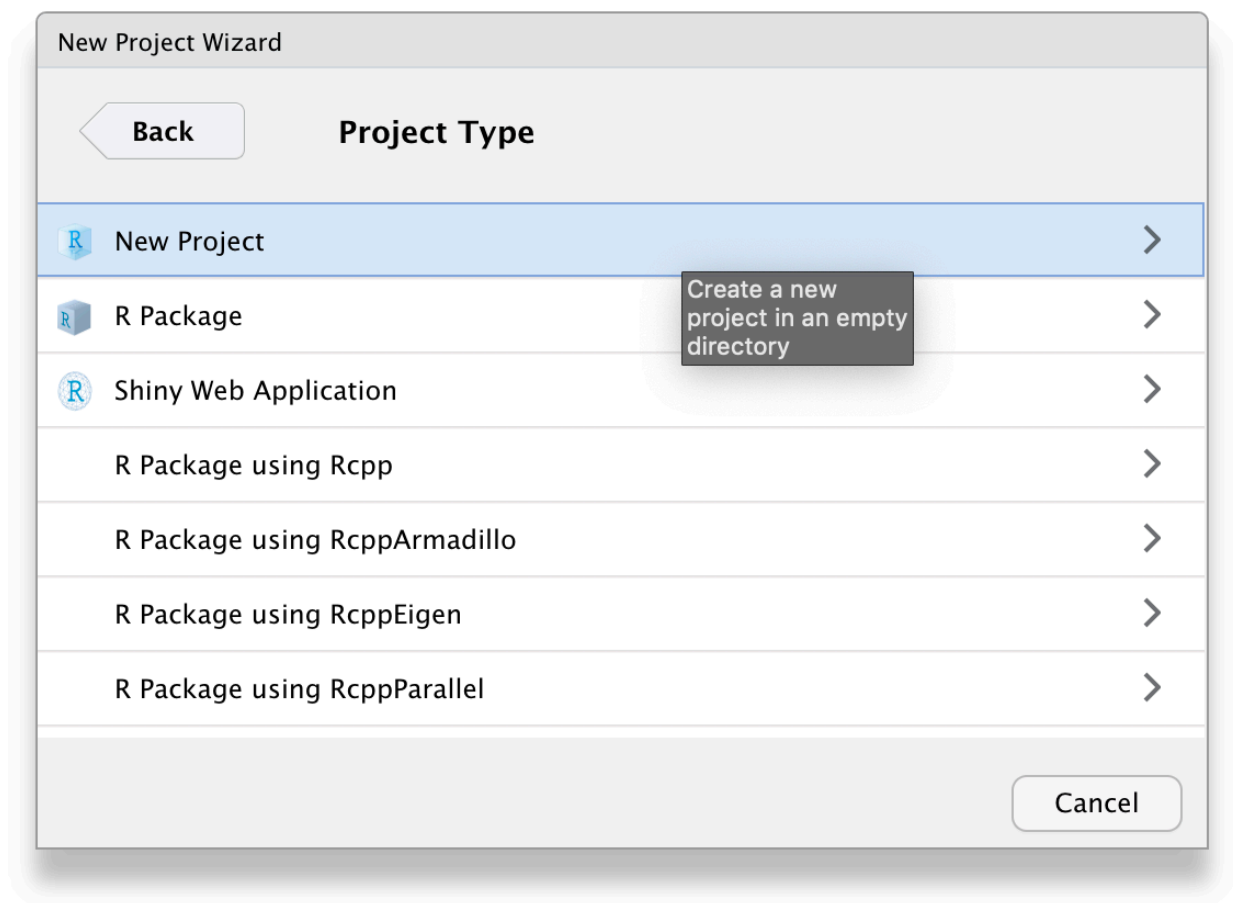
Figure 2.2: Starting a new project.
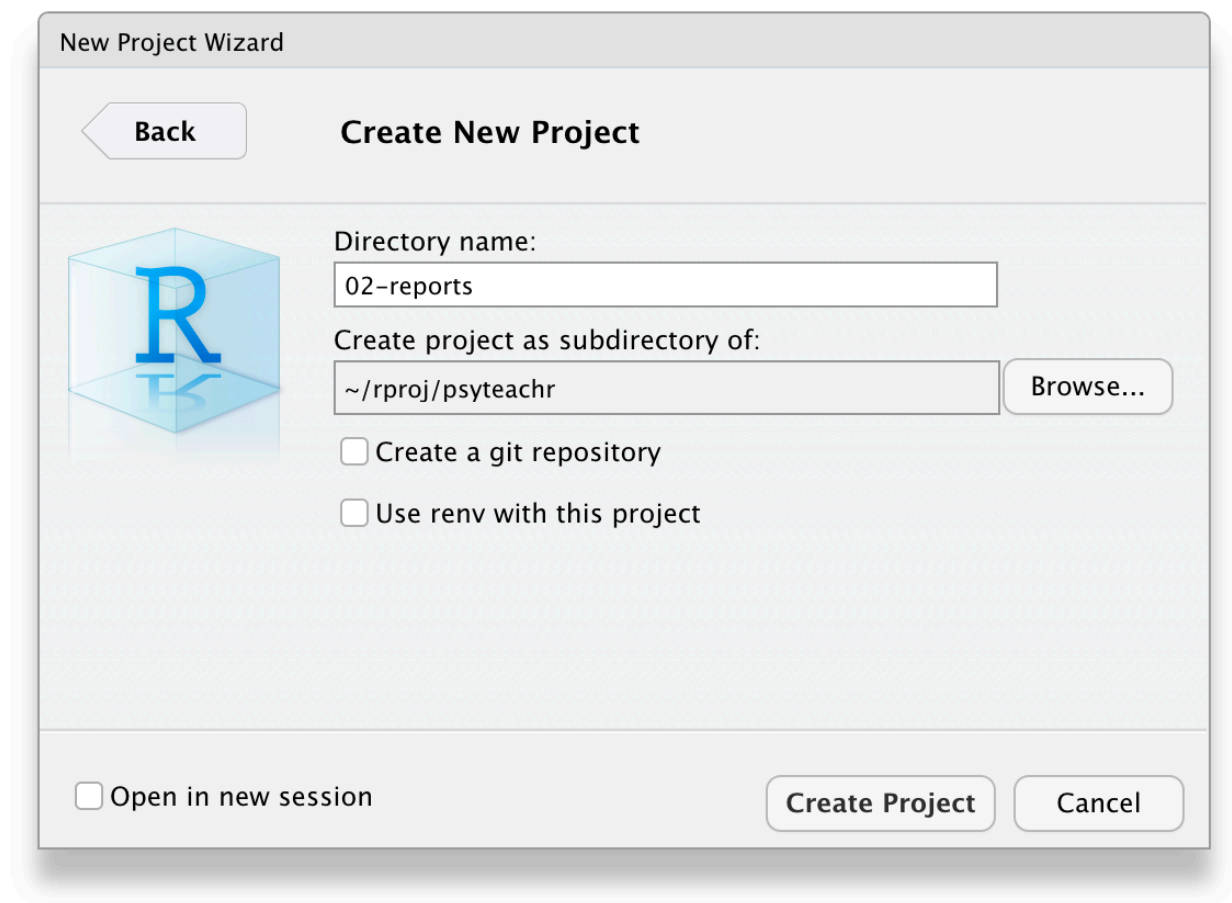
Figure 2.3: Starting a new project.

> Depending on your settings, you may also see a directory called `.Rproj.user`, which contains your specific user settings. You can ignore this and other "invisible" files that start with a full stop.

2.5   R Markdown

In this lesson, we will learn to make an R Markdown document with a table of contents, appropriate headers, code chunks, tables, images, inline R, and a bibliography.

We will use R Markdown to create reproducible reports, which enables mixing of text and code. A reproducible script will contain sections of code in code blocks. A code block starts and ends with three backtick symbols in a row, with some information about the code between curly brackets, such as `{r chunk-name, echo=FALSE}` (this runs the code, but does not show the text of the code block in the compiled document). The text outside of code blocks is written in markdown, which is a way to specify formatting, such as headers, paragraphs, lists, bolding, and links.

If you open up a new R Markdown file from a template, you will see an example document with several code blocks in it. To create an HTML or PDF report from an R Markdown (Rmd) document, you compile it. Compiling a document is called knitting in RStudio. There is a button that looks like a ball of yarn with needles through it that you click on to compile your file into a report.

> Create a new R Markdown file from the **File > New File > R Markdown...** menu. Change the title and author, then click the knit button to create an html file.

2.5.1   YAML Header

The YAML header is where you can set several options.

```
---
title: "My Demo Document"
author: "Me"
output:
  html_document:
    df_print: kable
    theme:
      version: 4
      bootswatch: yeti
    highlight: tango
    toc: true
    toc_float:
      collapsed: false
      smooth_scroll: false
    toc_depth: 3
    number_sections: false
---
```

> Try changing the values from `false` to `true` to see what the options do.

The `df_print: kable` option prints data frames using `knitr::kable`. You'll learn below how to further customise tables.

```
 8 ```{r setup, include=FALSE}
 9 library(tidyverse)
10 ```
11
12 ## Simulate data
13
14 Here we will simulate data from a study with two conditions.
   The mean in condition A is 0 and the mean in condition B is 1.
15
16 ```{r}
17 n <- 100
18
19 data <- data.frame(
20   id = 1:n,
21   dv = c(rnorm(n/2, 0), rnorm(n/2, 1)),
22   condition = rep(c("A", "B"), each = n/2)
23 )
24 ```
25
26 ## Plot data
27
28 ```{r}
29 ggplot(data, aes(condition, dv)) +
30   geom_violin(trim = FALSE) +
31   geom_boxplot(width = 0.25,
32               aes(fill = condition),
33               show.legend = FALSE)
34 ```
```

Figure 2.4: A reproducible script.

The built-in bootswatch themes are: default, cerulean, cosmo, darkly, flatly, journal, lumen, paper, readable, sandstone, simplex, spacelab, united, and yeti. You can view and download more themes.

Figure 2.5: Light themes in versions 3 and 4.

2.5.2   Setup

When you create a new R Markdown file in RStudio, a setup chunk is automatically created.

```
knitr::opts_chunk$set(echo = TRUE)
```

You can set more default options for code chunks here. See the knitr options documentation for explanations of the possible options.

```
knitr::opts_chunk$set(
  fig.width  = 8,
  fig.height = 5,
  fig.path   = 'images/',
  echo       = FALSE,
  warning    = TRUE,
  message    = FALSE,
  cache      = FALSE
)
```

The code above sets the following options:

- `fig.width  = 8` : default figure width is 8 inches (you can change this for individual figures)
- `fig.height = 5` : default figure height is 5 inches

- `fig.path`   = `'images/'` : figures are saved in the directory "images"
- `echo`   = `FALSE` : do not show code chunks in the rendered document
- `warning`   = `FALSE` : do not show any function warnings
- `message`   = `FALSE` : do not show any function messages
- `cache`   = `FALSE` : run all the code to create all of the images and objects each time you knit (set to `TRUE` if you have time-consuming code)

Find a list of the current chunk options by typing str(knitr::opts_chunk$get()) in the console.

You can also add the packages you need in this chunk using library(). Often when you are working on a script, you will realize that you need to load another add-on package. Don't bury the call to library(package_I_need) way down in the script. Put it in the top, so the user has an overview of what packages are needed.

> We'll be using function from the package tidyverse, so load that in your setup chunk.

2.5.3   Structure

If you include a table of contents (`toc`), it is created from your document headers. Headers in markdown are created by prefacing the header title with one or more hashes (`#`).

Use the following structure when developing your own analysis scripts:

- load in any add-on packages you need to use
- define any custom functions
- load or simulate the data you will be working with
- work with the data
- save anything you need to save

> Delete the default text and add some structure to your document by creating headers and subheaders. We're going to load some data, create a summary table, plot the data, and analyse it.

2.5.4   Code Chunks

You can include code chunks that create and display images, tables, or computations to include in your text. Let's start by loading some data.

First, create a code chunk in your document. This code loads some data from the web.

```r
pets <- read_csv("https://psyteachr.github.io/reprores/data/pets.csv")
```

```
## Rows: 800 Columns: 6

## -- Column specification --------------------------------------------------------
## Delimiter: ","
## chr (3): id, pet, country
## dbl (3): score, age, weight

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

2.5.4.1 Comments

You can add comments inside R chunks with the hash symbol (`#`). The R interpreter will ignore characters from the hash to the end of the line.

```r
# simulating new data

n <- nrow(pets) # the total number of pet
mu <- mean(pets$score) # the mean score for all pets
sd <- sd(pets$score) # the SD for score for all pets

simulated_scores <- rnorm(n, mu, sd)
```

It's usually good practice to start a code chunk with a comment that explains what you're doing there, especially if the code is not explained in the text of the report.

If you name your objects clearly, you often don't need to add clarifying comments. For example, if I'd named the three objects above `total_pet_n`, `mean_score` and `sd_score`, I would omit the comments.

It's a bit of an art to comment your code well. The best way to develop this skill is to read a lot of other people's code and have others review your code.

2.5.4.2 Tables

Next, create a code chunk where you want to display a table of the descriptives (e.g., Participants section of the Methods). We'll use tidyverse functions you will learn in the data wrangling lectures to create summary statistics for each group.

```r
pets %>%
  group_by(pet) %>%
  summarise(
    n = n(),
    mean_weight = mean(weight),
    mean_score = mean(score)
  )
```

```
## # A tibble: 3 x 4
##   pet        n mean_weight mean_score
##   <chr>  <int>       <dbl>      <dbl>
## 1 cat      300        9.37       90.2
## 2 dog      400       19.1       100.
## 3 ferret   100        4.78      112.
```

The table above is OK, but it could be more reader-friendly by changing the column labels, rounding the means, and adding a caption. You can use knitr::kable() for this.

```r
summary_table <-pets %>%
  group_by(pet) %>%
  summarise(
    n = n(),
    mean_weight = mean(weight),
    mean_score = mean(score)
  )
```

Table 2.2: Summary statistics for the pets dataset.

Pet Type	N	Mean Weight	Mean Score
cat	300	9.37	90.24
dog	400	19.07	99.98
ferret	100	4.78	111.78

```r
newnames <- c("Pet Type", "N", "Mean Weight", "Mean Score")

knitr::kable(summary_table,
             digits = 2,
             col.names = newnames,
             caption = "Summary statistics for the pets dataset.")
```

> Notice that the r chunk specifies the option `results='asis'`. This lets you format the table using the `kable()` function from `knitr`. You can also use more specialised functions from papaja or kableExtra to format your tables.

2.5.4.3 Images

Next, create a code chunk where you want to display an image in your document. Let's put it in the Results section. We'll use some code that you'll learn more about in the data visualisation lecture to show violin-boxplots for the groups.

Notice how the figure caption is formatted in the chunk options.

```r
ggplot(pets, aes(pet, score, fill = country)) +
  geom_violin(alpha = 0.5) +
  geom_boxplot(width = 0.25,
               position = position_dodge(width = 0.9),
               show.legend = FALSE) +
  scale_fill_manual(values = c("orange", "dodgerblue")) +
  xlab("") +
  ylab("Score") +
  theme(text = element_text(size = 20, family = "Times"))
```

```
\begin{figure}

{\centering \includegraphics[width=1\linewidth]{02-repro_files/figure-latex/unnamed-chunk-7-1}

}

\caption{Figure 1. Scores by pet type and country.}\label{fig:unnamed-chunk-7}
\end{figure}
```
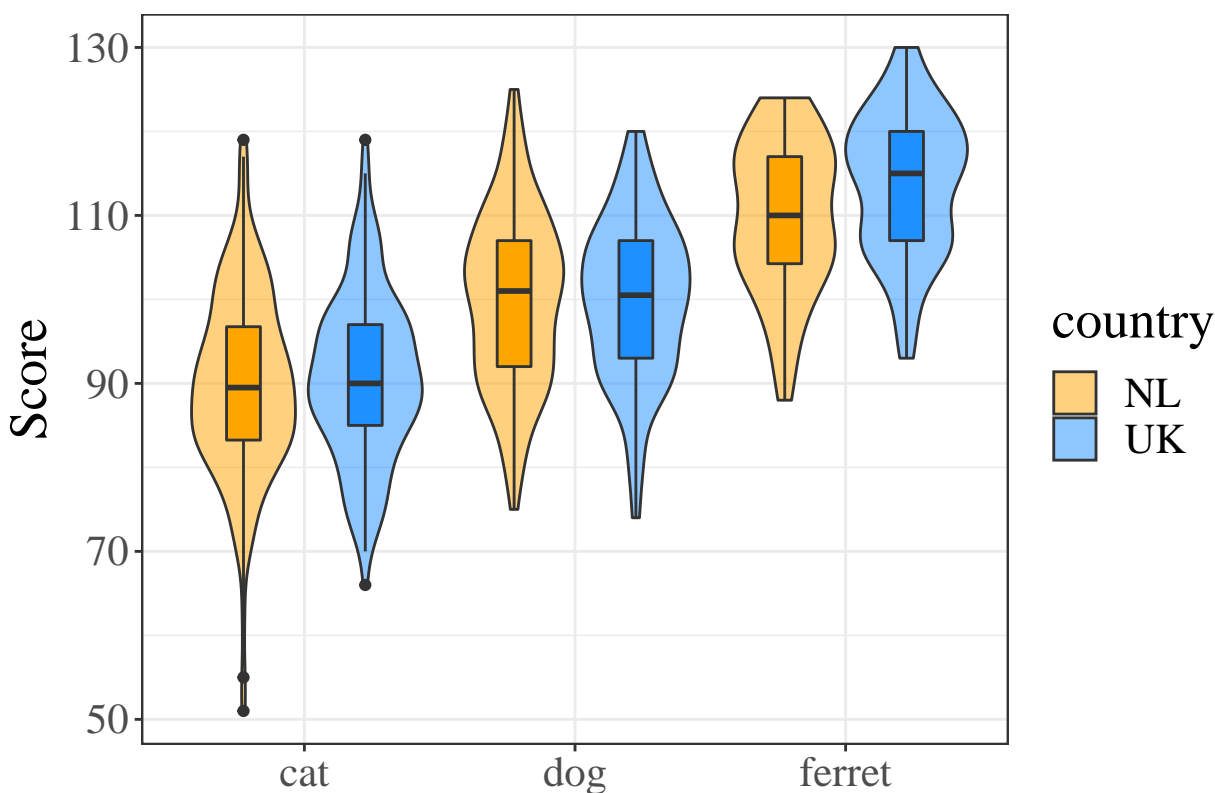
Figure 2.6: Figure 1. Scores by pet type and country.

> The last line changes the default text size and font, which can be useful for generating figures that meet a journal's requirements.

You can also include images that you did not create in R using the typical markdown syntax for images:

```
![All the Things by [Hyperbole and a Half](http://hyperboleandahalf.blogspot.com/)](images/memes/x-all-
```

2.5.4.4 In-line R

Now let's analyse the pets data to see if cats are heavier than ferrets. First we'll run the analysis code. Then we'll save any numbers we might want to use in our manuscript to variables and round them appropriately. Finally, we'll use glue::glue() to format a results string.

```r
# analysis
cat_weight <- filter(pets, pet == "cat") %>% pull(weight)
ferret_weight <- filter(pets, pet == "ferret") %>% pull(weight)
weight_test <- t.test(cat_weight, ferret_weight)

# round individual values you want to report
t <- weight_test$statistic %>% round(2)
df <- weight_test$parameter %>% round(1)
p <- weight_test$p.value %>% round(3)
# handle p-values < .001
```

Figure 2.7: All the Things by Hyperbole and a Half

```r
p_symbol <- ifelse(p < .001, "<", "=")
if (p < .001) p <- .001

# format the results string
weight_result <- glue::glue("t = {t}, df = {df}, p {p_symbol} {p}")
```

You can insert the results into a paragraph with inline R code that looks like this:

Rendered text:
Cats were significantly heavier than ferrets (t = 18.42, df = 180.4, p < 0.001).

2.5.5   Bibliography

There are several ways to do in-text citations and automatically generate a bibliography in R Markdown. Markdown files need a BibTex file that contains the references you need to cite. You specify the name of this file in the YAML header, like `bibliography: filename.bib` and reference citations in text using an at symbol and a shortname, like `[@tidyverse]`.

2.5.5.1   Converting from reference software

Most reference software like EndNote, Zotero or Mendeley have exporting options that can export to BibTeX format. You just need to check the shortnames in the resulting file.

2.5.5.2   Create a BibTeX File Manually

You can make a BibTeX file and add citations manually. Make a new Text File in RStudio called "bibliography.bib".

Next, add the line `bibliography: bibliography.bib` to your YAML header.

You can add citations in the following format:

```
@article{shortname,
  author = {Author One and Author Two and Author Three},
  title = {Paper Title},
  journal = {Journal Title},
  volume = {vol},
  number = {issue},
  pages = {startpage--endpage},
  year = {year},
  doi = {doi}
}
```

2.5.5.3   Citing R packages

You can get the citation for an R package using the functions `citation()` and `toBibtex()`. You can paste the bibtex entry into your bibliography.bib file. Make sure to add a short name (e.g., "faux") before the first comma to refer to the reference.

```r
citation(package="faux") %>% toBibtex()
```

```
## @Manual{,
##   title = {faux: Simulation for Factorial Designs},
##   author = {Lisa DeBruine},
##   doi = {10.5281/zenodo.2669586},
##   publisher = {Zenodo},
##   year = {2021},
##   note = {R package version 1.1.0},
##   url = {https://debruine.github.io/faux/},
## }
```

2.5.5.4   Download Citation Info

Google Scholar entries have a BibTeX citation option. This is usually the easiest way to get the relevant
values, although you have to add the DOI yourself.

Some journal websites also let you download citations in bibtex format. For example, go to the publisher's
page for Equivalence Testing for Psychological Research: A Tutorial, click on the Cite button (in the sidebar
or under the bottom Explore More menu), choose BibTeX format, and download the citation. You can open
up the file in a text editor and copy the text. It should look like this:

```
@article{doi:10.1177/2515245918770963,
author = {Daniël Lakens and Anne M. Scheel and Peder M. Isager},
title ={Equivalence Testing for Psychological Research: A Tutorial},
journal = {Advances in Methods and Practices in Psychological Science},
volume = {1},
number = {2},
pages = {259-269},
year = {2018},
doi = {10.1177/2515245918770963},

URL = {
        https://doi.org/10.1177/2515245918770963

},
eprint = {
        https://doi.org/10.1177/2515245918770963

}
,
    abstract = { Psychologists must be able to test both for the presence of an effect and for the absen
}
```

Paste the reference into your bibliography.bib file. Change `doi:10.1177/2515245918770963` in the first line
of the reference to a shortname you will use to cite the reference in your manuscript. We'll use `TOSTtutorial`.

2.5.5.5   In-text citations

You can cite reference in text like this:

```
This tutorial uses several R packages [@tidyverse;@rmarkdown].
```

This tutorial uses several R packages (**??**).

Put a minus in front of the @ if you just want the year:

```
Lakens, Scheel and Isengar [-@TOSTtutorial] wrote a tutorial explaining how to test for the absence of a
```

Lakens, Scheel and Isengar (**?**) wrote a tutorial explaining how to test for the absence of an effect.

2.5.5.6 Citation Styles

You can search a list of style files for various journals and download a file that will format your bibliography for a specific journal's style. You'll need to add the line `csl: filename.csl` to your YAML header.

> Add some citations to your bibliography.bib file, reference them in your text, and render your manuscript to see the automatically generated reference section. Try a few different citation style files.

2.5.6 Output Formats

You can knit your file to PDF or Word if you have the right packages installed on your computer. You can also create presentations, dashboards, websites, and even books with R markdown. In fact, the book you are reading right now was created using R markdown. See RStudio Formats for a list of all the output types.

2.6 Glossary

term	definition
absolute-path	A file path that starts with / and is not appended to the working directory
chunk	A section of code in an R Markdown file
directory	A collection or "folder" of files on a computer.
extension	The end part of a file name that tells you what type of file it is (e.g., .R or .Rmd).
knit	To create an HTML, PDF, or Word document from an R Markdown (Rmd) document
markdown	A way to specify formatting, such as headers, paragraphs, lists, bolding, and links.
path	A string representing the location of a file or directory.
project	A way to organise related files in RStudio
r-markdown	The R-specific version of markdown: a way to specify formatting, such as headers, paragraphs, lists, b
relative-path	The location of a file in relation to the working directory.
reproducibility	The extent to which the findings of a study can be repeated in some other context
working-directory	The filepath where R is currently reading and writing files.
yaml	A structured format for information

2.7 Further Resources

- Chapter 27: R Markdown in *R for Data Science*
- R Markdown Cheat Sheet
- R Markdown reference Guide
- R Markdown Tutorial
- R Markdown: The Definitive Guide by Yihui Xie, J. J. Allaire, & Garrett Grolemund
- Project Structure by Danielle Navarro
- How to name files by Jenny Bryan
- Papaja Reproducible APA Manuscripts
- The Turing Way

Chapter 3

Data Visualisation

3.1 Learning Objectives

Basic

1. Understand what types of graphs are best for different types of data (video)

   - 1 discrete
   - 1 continuous
   - 2 discrete
   - 2 continuous
   - 1 discrete, 1 continuous
   - 3 continuous

2. Create common types of graphs with ggplot2 (video)

   - `geom_bar()`
   - `geom_density()`
   - `geom_freqpoly()`
   - `geom_histogram()`
   - `geom_col()`
   - `geom_boxplot()`
   - `geom_violin()`
   - Vertical Intervals
     - `geom_crossbar()`
     - `geom_errorbar()`
     - `geom_linerange()`
     - `geom_pointrange()`
   - `geom_point()`
   - `geom_smooth()`

3. Set custom size, labels, colours, and themes (video)
4. Combine plots on the same plot, as facets, or as a grid using patchwork (video)
5. Save plots as an image file (video)

Intermediate

6. Add lines to graphs
7. Deal with overlapping data

8. Create less common types of graphs

   - `geom_tile()`
   - `geom_density2d()`
   - `geom_bin2d()`
   - `geom_hex()`
   - `geom_count()`

9. Adjust axes (e.g., flip coordinates, set axis limits)
10. Create interactive graphs with `plotly`

3.2 Setup

```
# libraries needed for these graphs
library(tidyverse)
library(patchwork)
library(plotly)
library(ggwordcloud)
library(reprores)


set.seed(30250) # makes sure random numbers are reproducible
```

3.3 Common Variable Combinations

Continuous variables are properties you can measure, like height. Discrete variables are things you can count, like the number of pets you have. Categorical variables can be nominal, where the categories don't really have an order, like cats, dogs and ferrets (even though ferrets are obviously best). They can also be ordinal, where there is a clear order, but the distance between the categories isn't something you could exactly equate, like points on a Likert rating scale.

Different types of visualisations are good for different types of variables.

Load the `pets` dataset and explore it with `glimpse(pets)` or `View(pets)`. This is a simulated dataset with one random factor (`id`), two categorical factors (`pet`, `country`) and three continuous variables (`score`, `age`, `weight`).

```
data("pets", package = "reprores")
# if you don't have the reprores package, use:
# pets <- read_csv("https://psyteachr.github.io/reprores/data/pets.csv", col_types = "cffiid")
glimpse(pets)


## Rows: 800
## Columns: 6
## $ id      <chr> "S001", "S002", "S003", "S004", "S005", "S006", "S007", "S008"~
## $ pet     <fct> dog, dog, dog, dog, dog, dog, dog, dog, dog, dog, dog, dog, do~
## $ country <fct> UK, UK, UK, UK, UK, UK, UK, UK, UK, UK, UK, UK, UK, UK, UK, UK~
## $ score   <int> 90, 107, 94, 120, 111, 110, 100, 107, 106, 109, 85, 110, 102, ~
## $ age     <int> 6, 8, 2, 10, 4, 8, 9, 8, 6, 11, 5, 9, 1, 10, 7, 8, 1, 8, 5, 13~
## $ weight  <dbl> 19.78932, 20.01422, 19.14863, 19.56953, 21.39259, 21.31880, 19~
```

> Before you read ahead, come up with an example of each type of variable combination and sketch the types of graphs that would best display these data.
>
> - 1 categorical
> - 1 continuous
> - 2 categorical
> - 2 continuous
> - 1 categorical, 1 continuous
> - 3 continuous

3.4   Basic Plots

R has some basic plotting functions, but they're difficult to use and aesthetically not very nice. They can be useful to have a quick look at data while you're working on a script, though. The function `plot()` usually defaults to a sensible type of plot, depending on whether the arguments `x` and `y` are categorical, continuous, or missing.

```
plot(x = pets$pet)
```
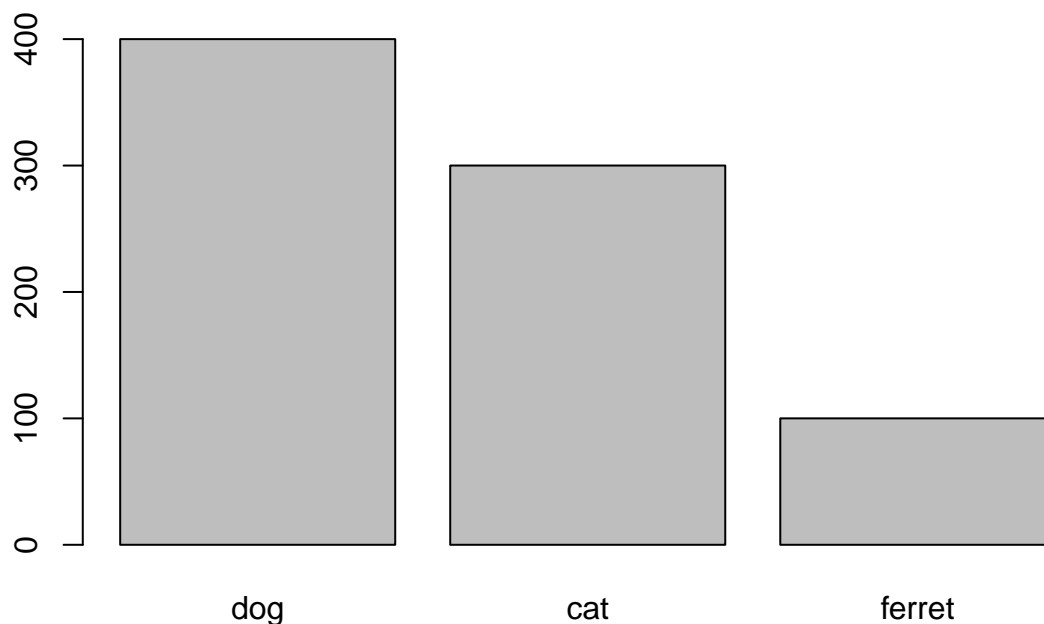

Figure 3.1: plot() with categorical x

```
plot(x = pets$pet, y = pets$score)
```

```
plot(x = pets$age, y = pets$weight)
```

The function `hist()` creates a quick histogram so you can see the distribution of your data. You can adjust how many columns are plotted with the argument `breaks`.
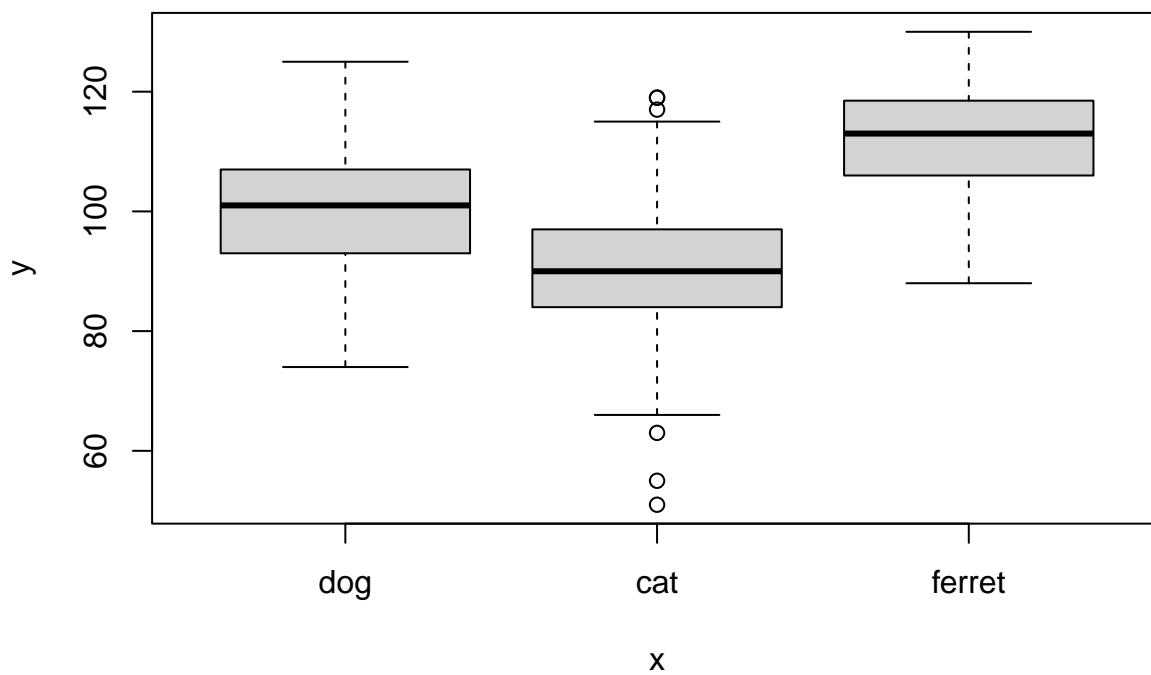
Figure 3.2: plot() with categorical x and continuous y
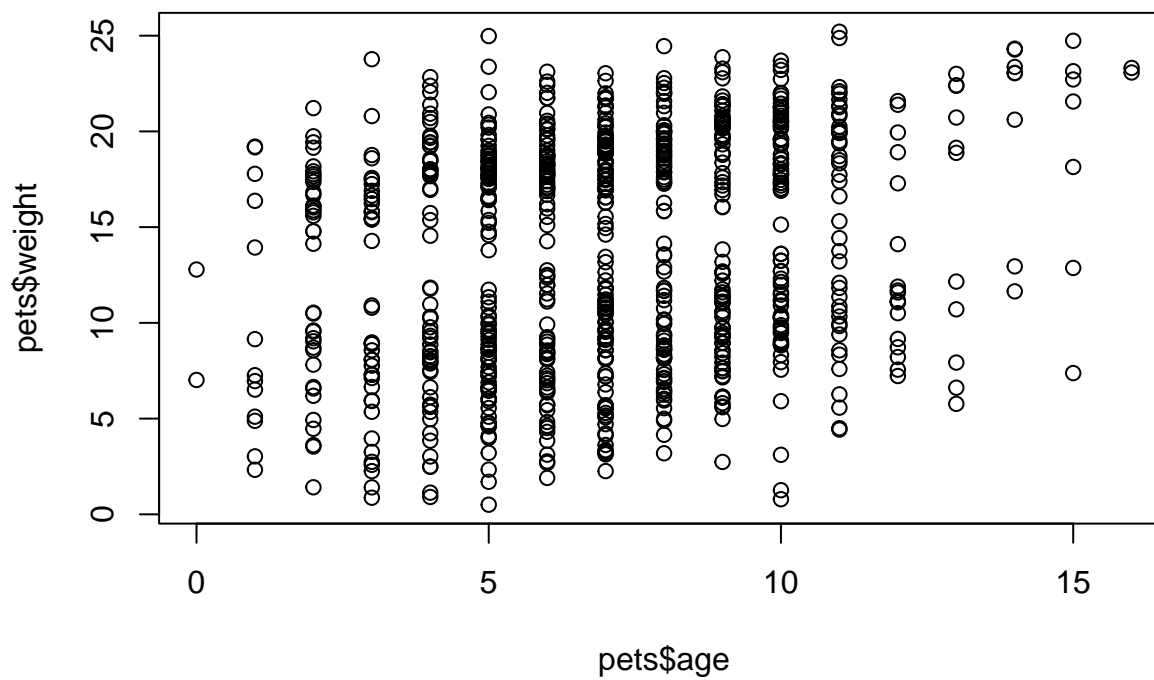
Figure 3.3: plot() with continuous x and y
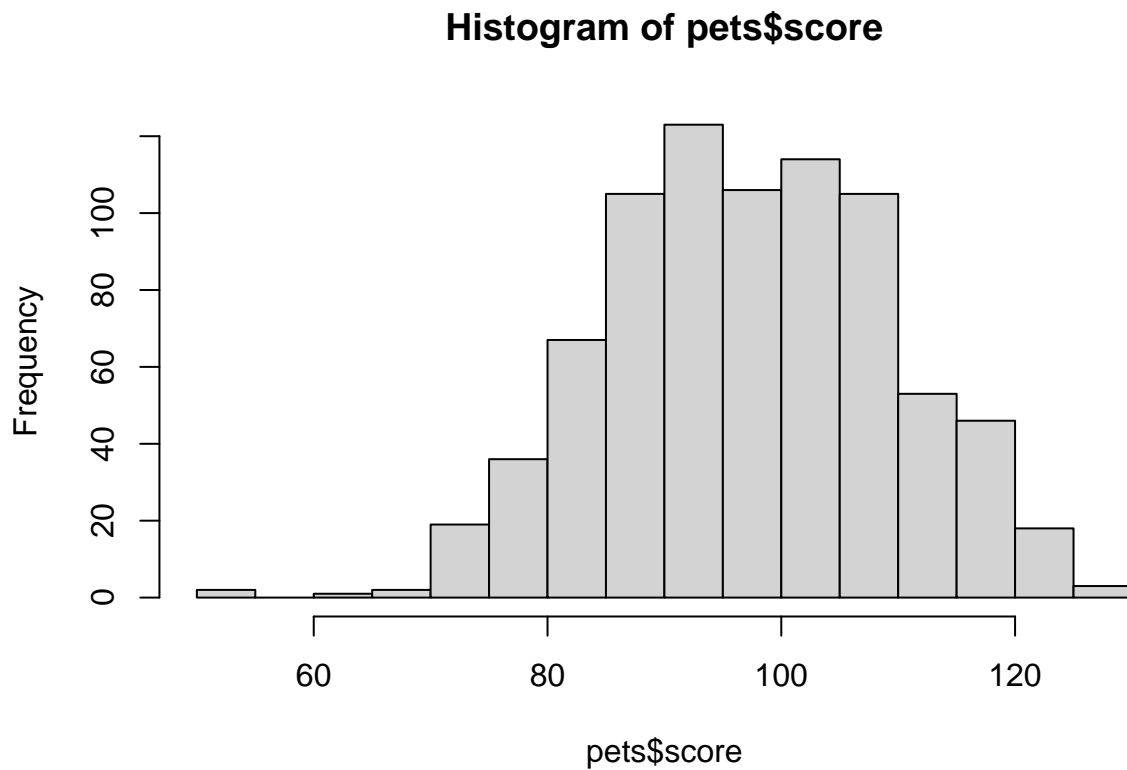
```
hist(pets$score, breaks = 20)
```

Histogram of pets$score

Figure 3.4: hist()

3.5 GGplots

While the functions above are nice for quick visualisations, it's hard to make pretty, publication-ready plots. The package `ggplot2` (loaded with `tidyverse`) is one of the most common packages for creating beautiful visualisations.

`ggplot2` creates plots using a "grammar of graphics" where you add geoms in layers. It can be complex to understand, but it's very powerful once you have a mental model of how it works.

Let's start with a totally empty plot layer created by the `ggplot()` function with no arguments.

```
ggplot()
```

The first argument to `ggplot()` is the `data` table you want to plot. Let's use the `pets` data we loaded above. The second argument is the `mapping` for which columns in your data table correspond to which properties of the plot, such as the x-axis, the y-axis, line `colour` or `linetype`, point `shape`, or object `fill`. These mappings are specified by the `aes()` function. Just adding this to the `ggplot` function creates the labels and ranges for the `x` and `y` axes. They usually have sensible default values, given your data, but we'll learn how to change them later.

```
mapping <- aes(x = pet,
               y = score,
```
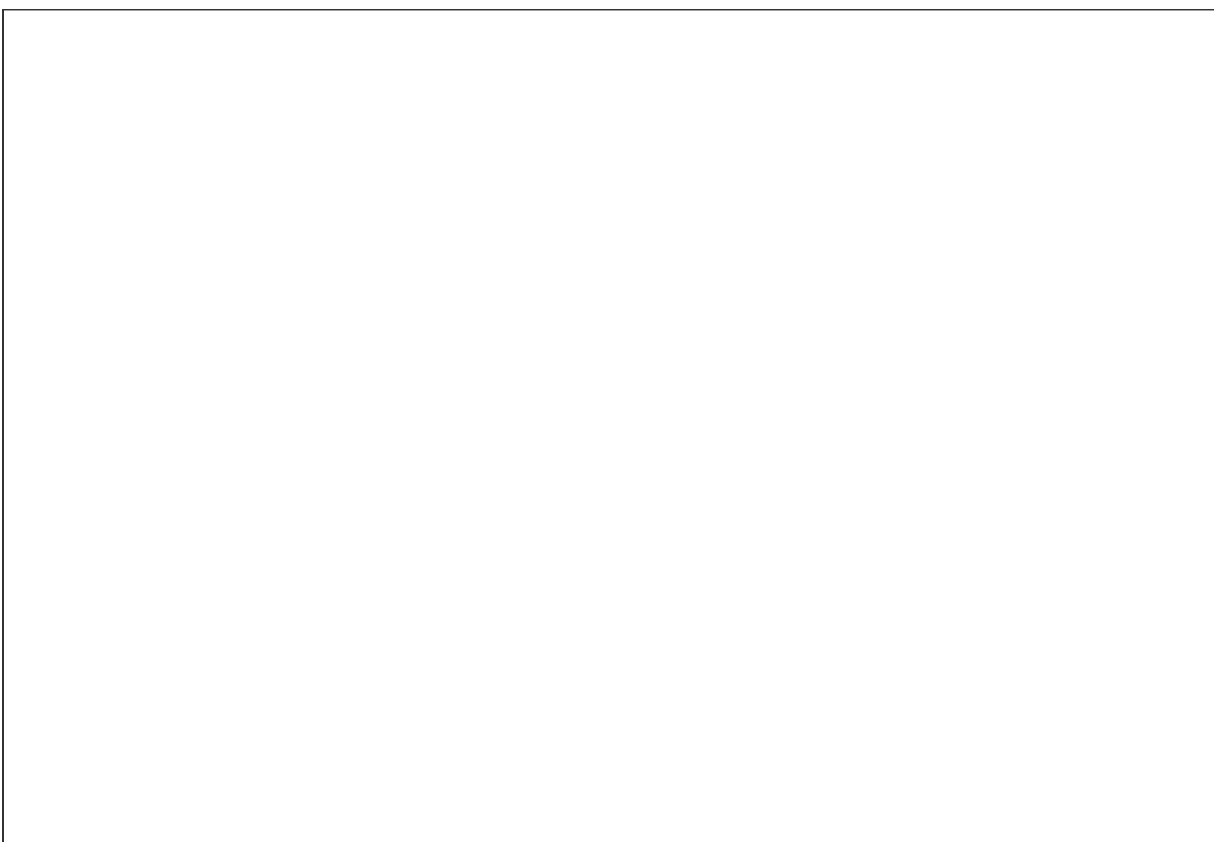
Figure 3.5: A plot base created by ggplot()

```
                colour = country,
                fill = country)
ggplot(data = pets, mapping = mapping)
```
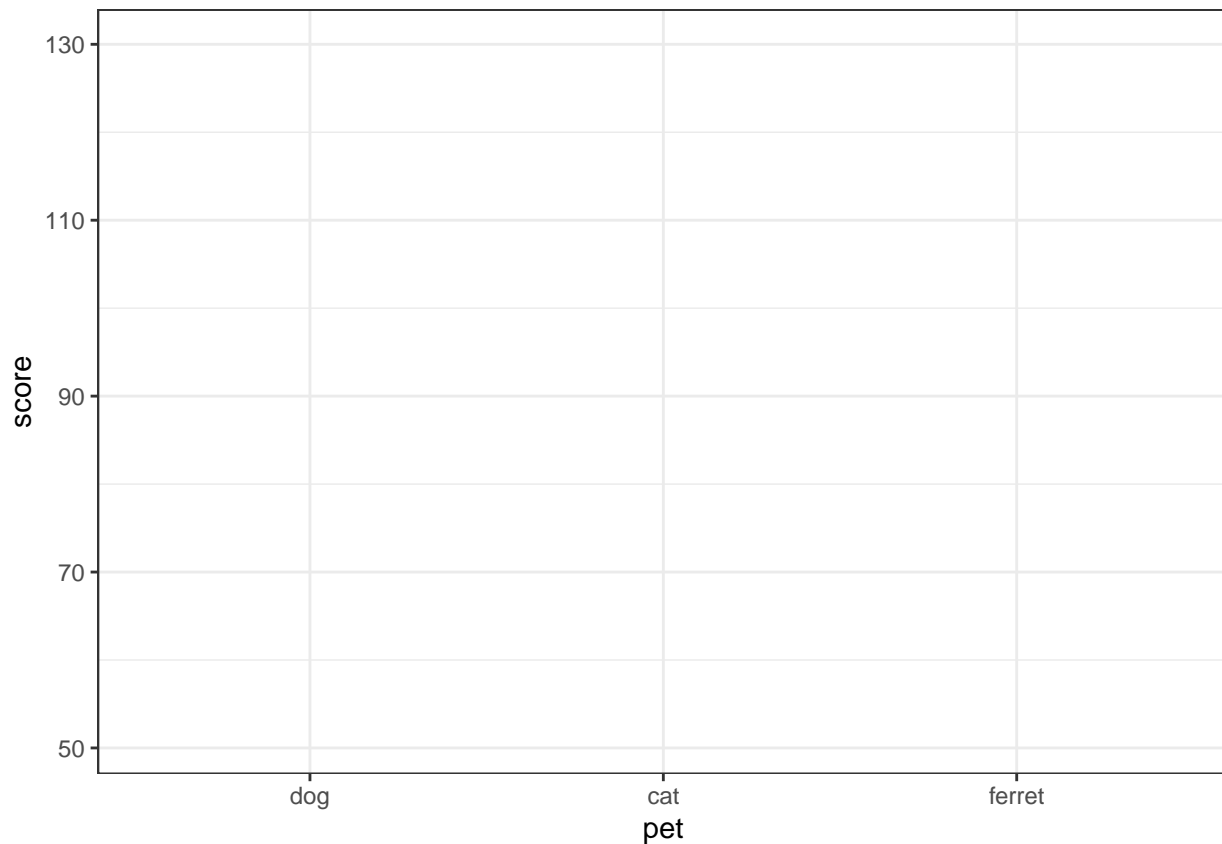

Figure 3.6: Empty ggplot with x and y labels

> People usually omit the argument names and just put the `aes()` function directly as the second
> argument to `ggplot`. They also usually omit `x` and `y` as argument names to `aes()` (but you have to
> name the other properties).

Next we can add "geoms", or plot styles. You literally add them with the `+` symbol. You can also add other
plot attributes, such as labels, or change the theme and base font size.

```
ggplot(pets, aes(pet, score, colour = country, fill = country)) +
  geom_violin(alpha = 0.5) +
  labs(x = "Pet type",
       y = "Score on an Important Test",
       colour = "Country of Origin",
       fill = "Country of Origin",
       title = "My first plot!") +
  theme_bw(base_size = 15)
```
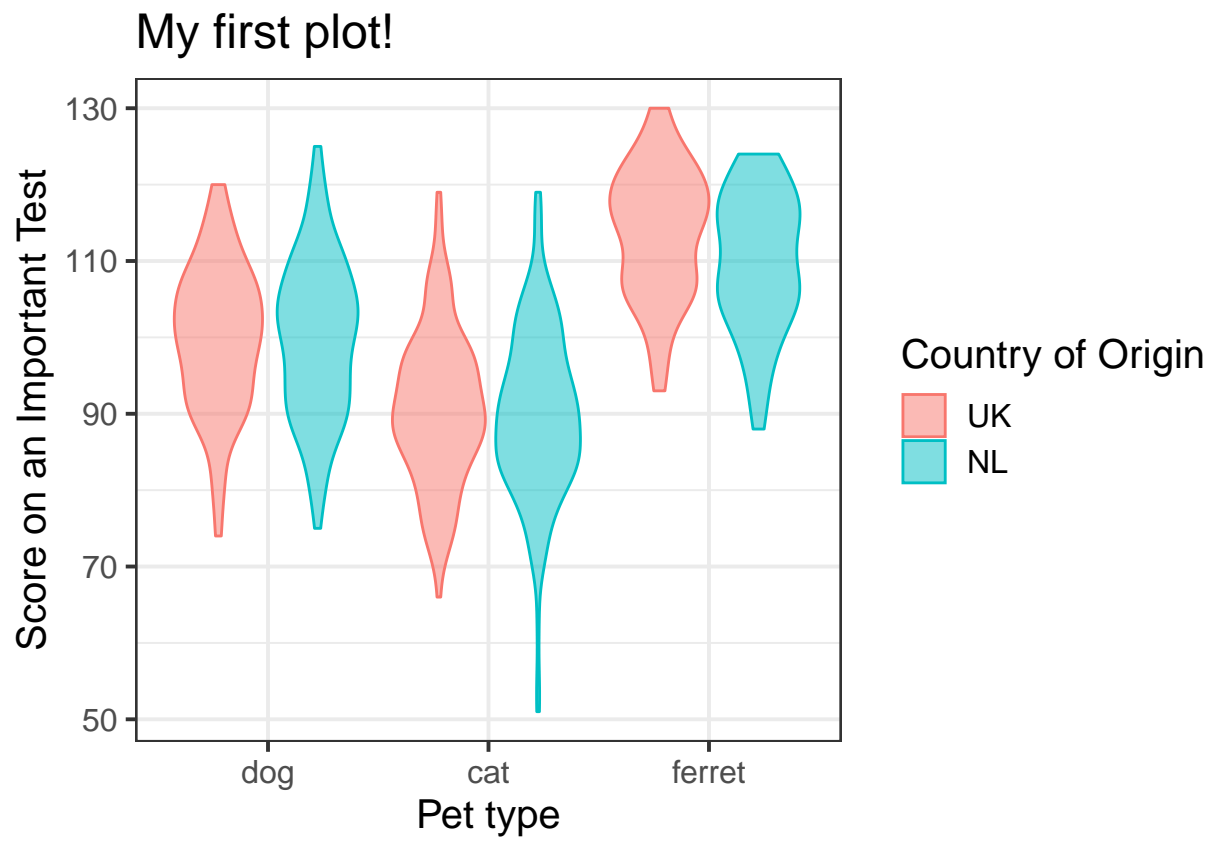
Figure 3.7: Violin plot with country represented by colour.

3.6   Common Plot Types

There are many geoms, and they can take different arguments to customise their appearance. We'll learn about some of the most common below.

3.6.1   Bar plot

Bar plots are good for categorical data where you want to represent the count.

```
ggplot(pets, aes(pet)) +
  geom_bar()
```
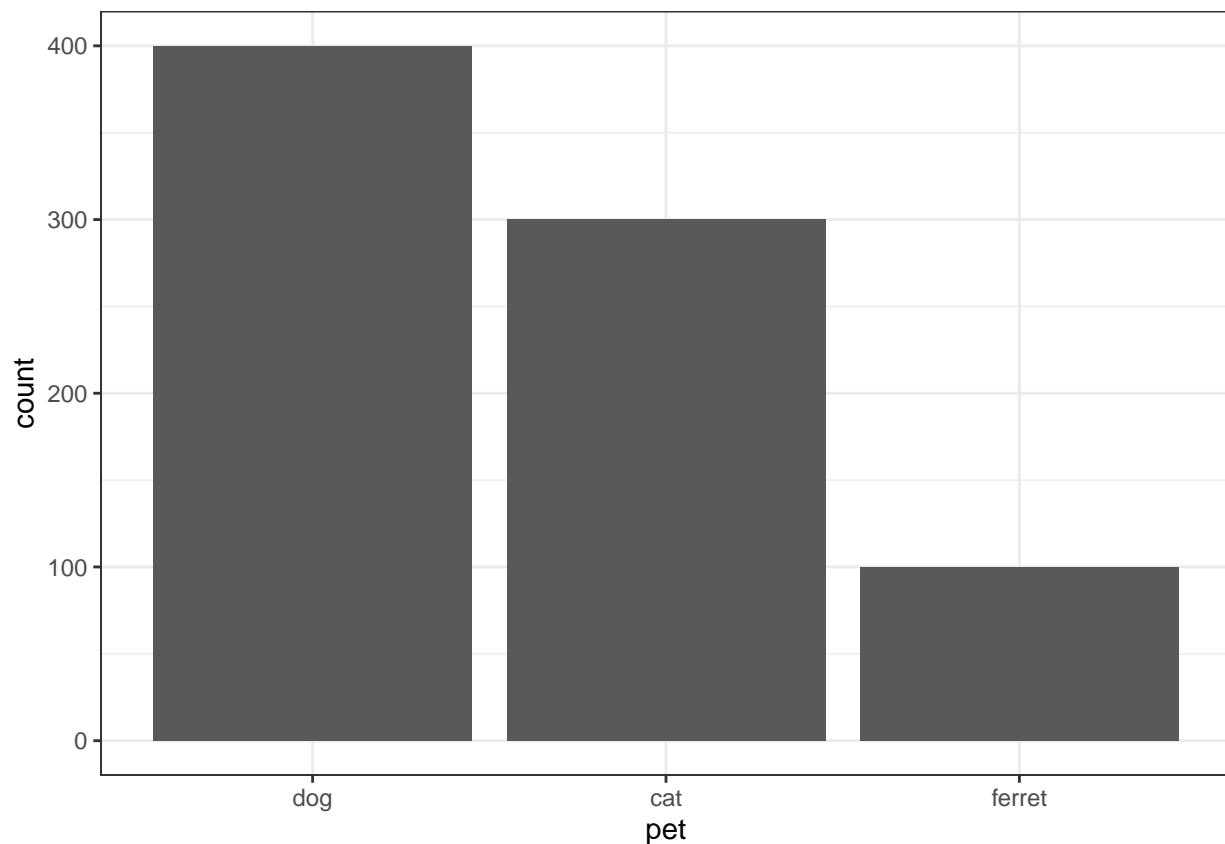

Figure 3.8: Bar plot

3.6.2   Density plot

Density plots are good for one continuous variable, but only if you have a fairly large number of observations.

```
ggplot(pets, aes(score)) +
  geom_density()
```

You can represent subsets of a variable by assigning the category variable to the argument `group`, `fill`, or `color`.
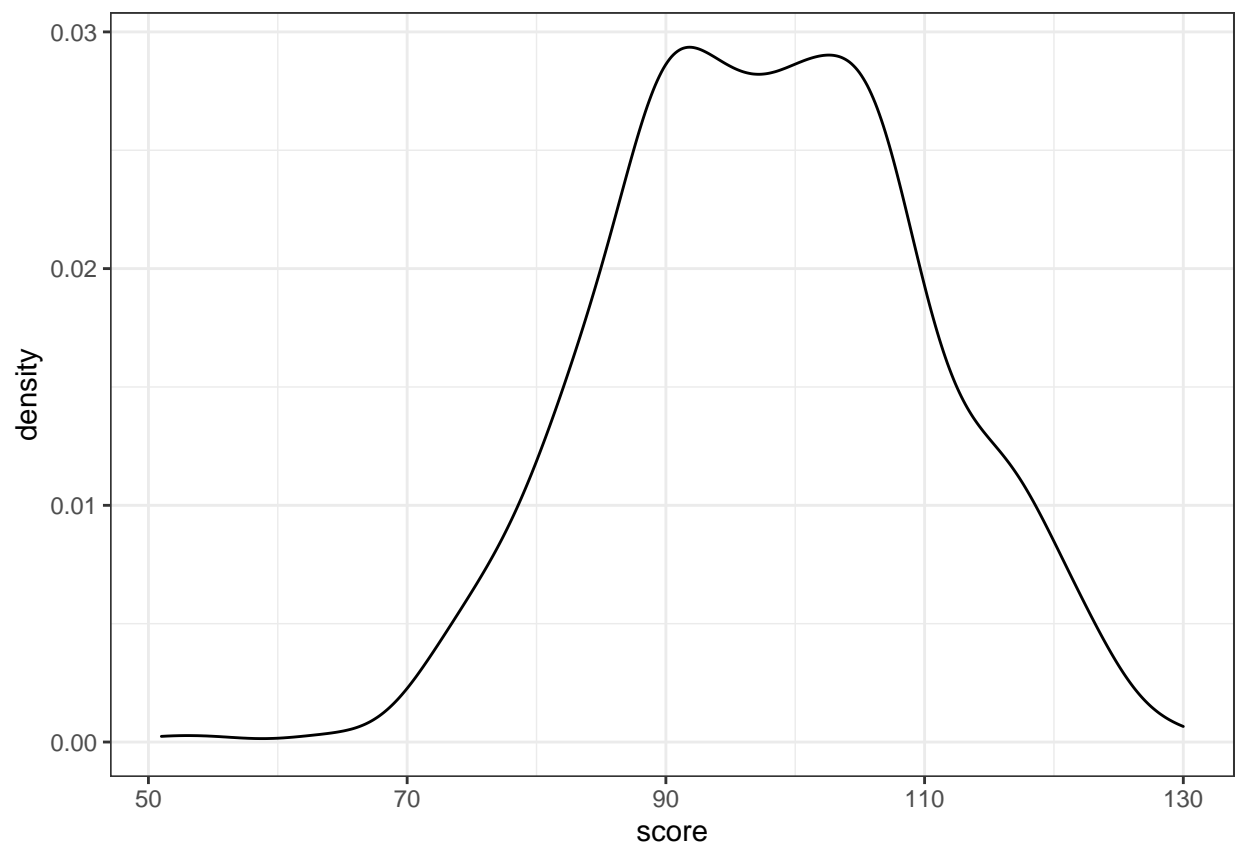
Figure 3.9: Density plot

```
ggplot(pets, aes(score, fill = pet)) +
  geom_density(alpha = 0.5)
```
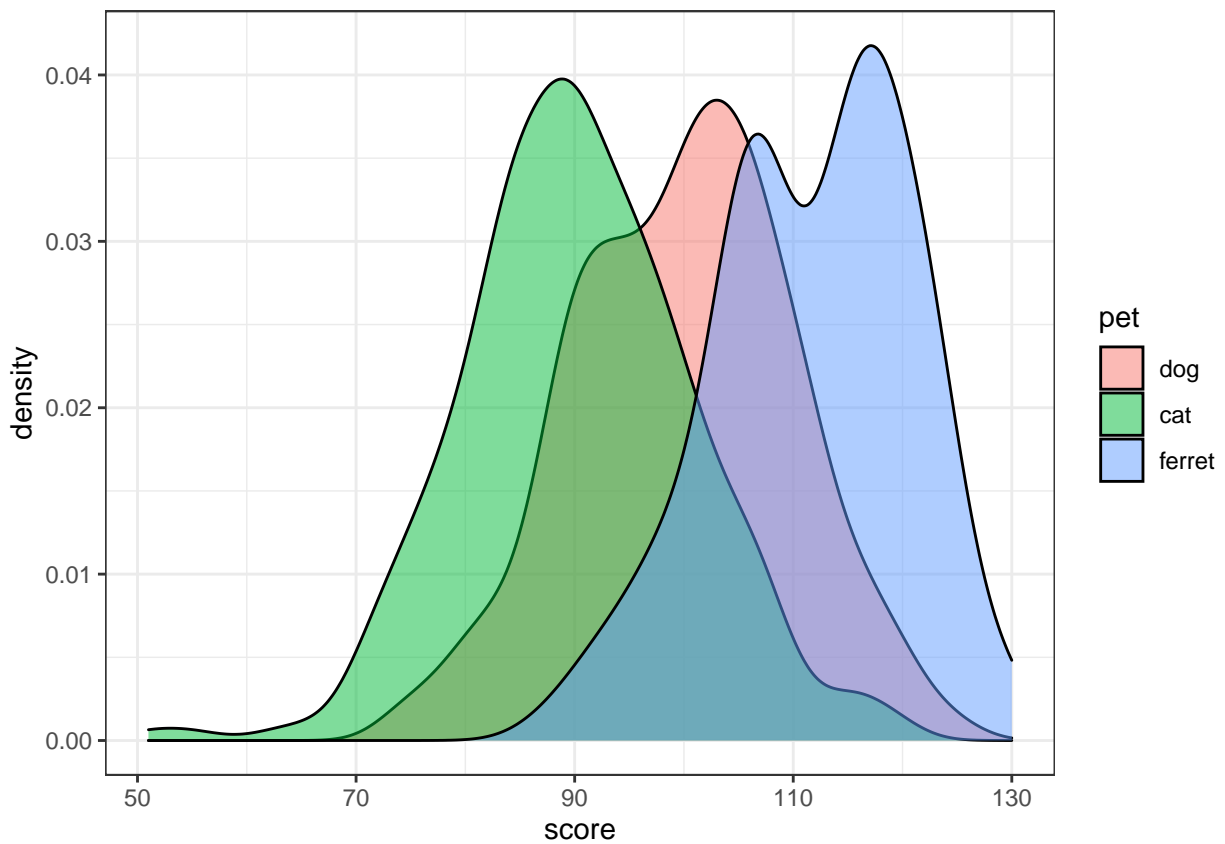

Figure 3.10: Grouped density plot

> Try changing the `alpha` argument to figure out what it does.

3.6.3 Frequency polygons

If you want the y-axis to represent count rather than density, try `geom_freqpoly()`.

```
ggplot(pets, aes(score, color = pet)) +
  geom_freqpoly(binwidth = 5)
```

> Try changing the `binwidth` argument to 10 and 1. How do you figure out the right value?

3.6.4 Histogram

Histograms are also good for one continuous variable, and work well if you don't have many observations. Set the `binwidth` to control how wide each bar is.
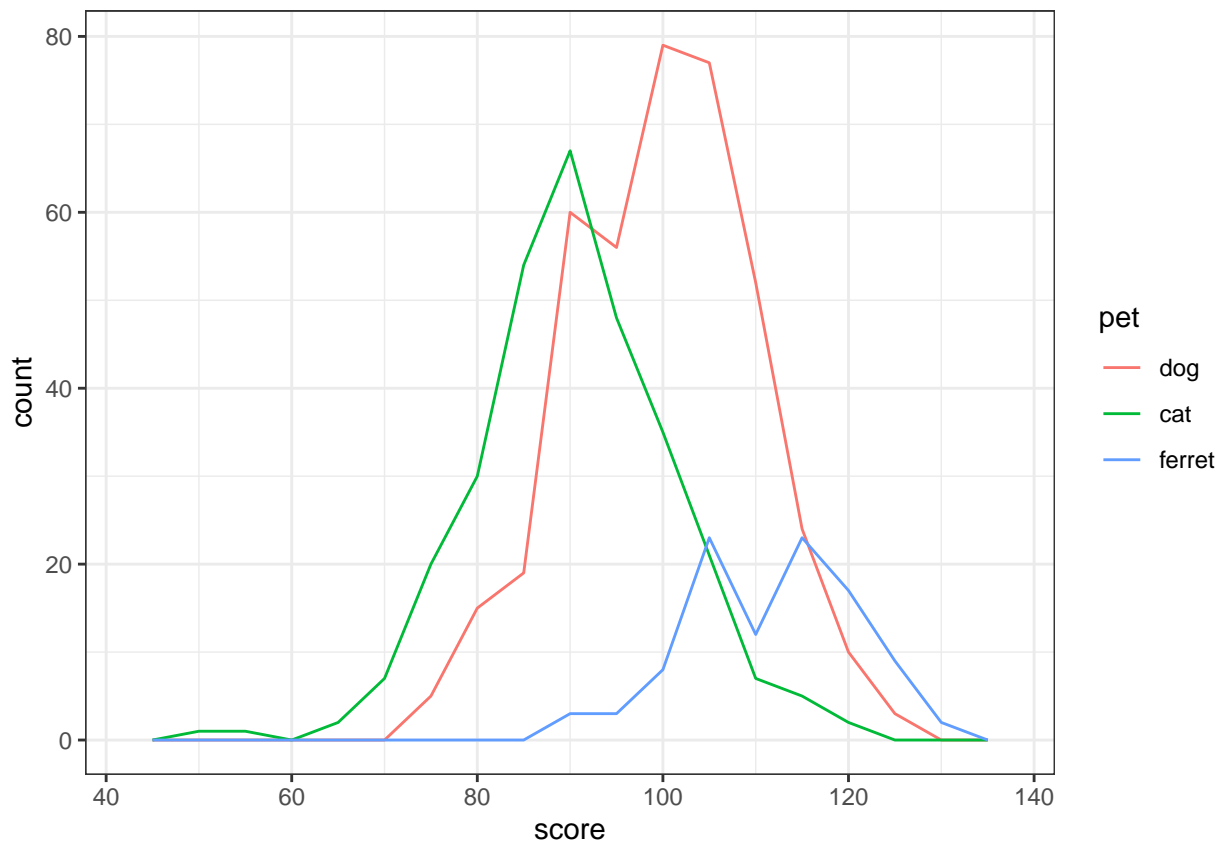
Figure 3.11: Frequency ploygon plot

```
ggplot(pets, aes(score)) +
  geom_histogram(binwidth = 5, fill = "white", color = "black")
```
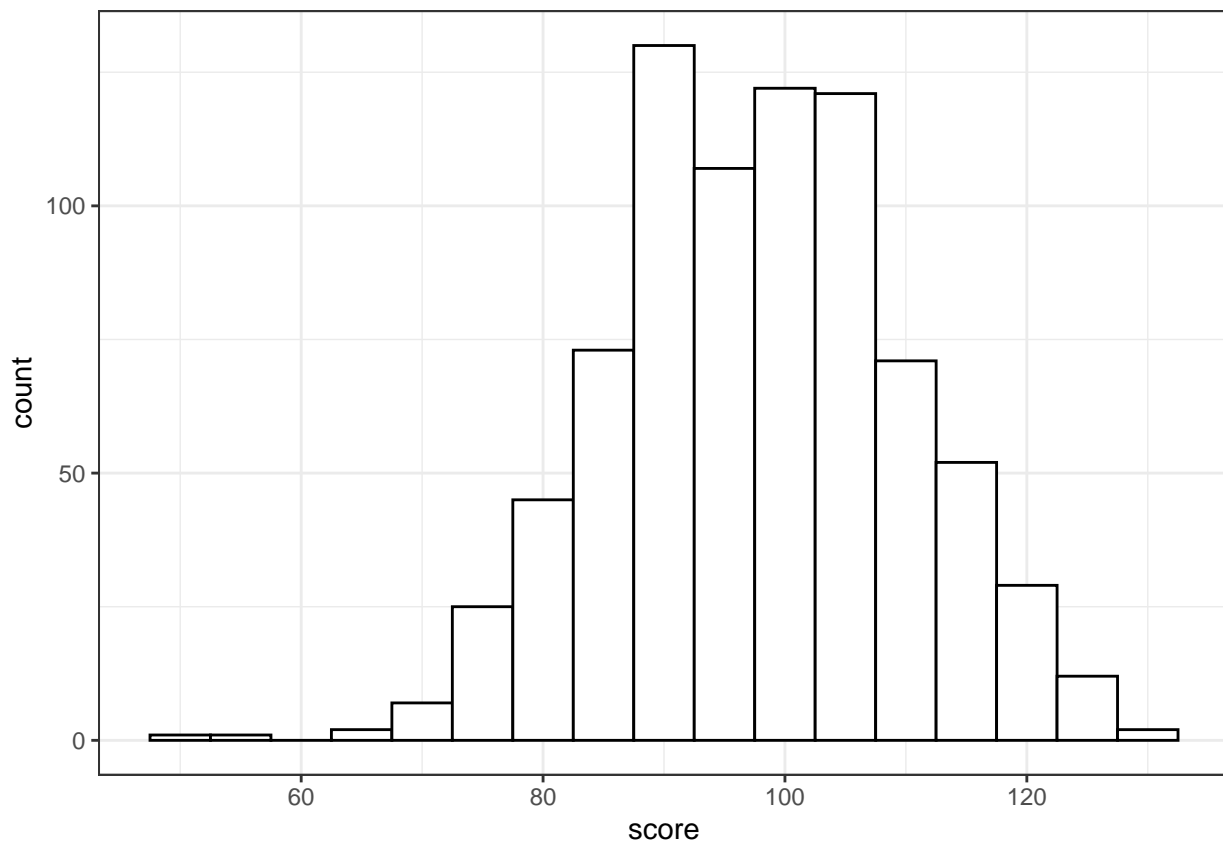

Figure 3.12: Histogram

> Histograms in ggplot look pretty bad unless you set the `fill` and `color`.

If you show grouped histograms, you also probably want to change the default `position` argument.

```
ggplot(pets, aes(score, fill=pet)) +
  geom_histogram(binwidth = 5, alpha = 0.5,
                 position = "dodge")
```

> Try changing the `position` argument to "identity", "fill", "dodge", or "stack".

3.6.5   Column plot

Column plots are the worst way to represent grouped continuous data, but also one of the most common. If your data are already aggregated (e.g., you have rows for each group with columns for the mean and standard error), you can use `geom_bar` or `geom_col` and `geom_errorbar` directly. If not, you can use the function `stat_summary` to calculate the mean and standard error and send those numbers to the appropriate geom for plotting.
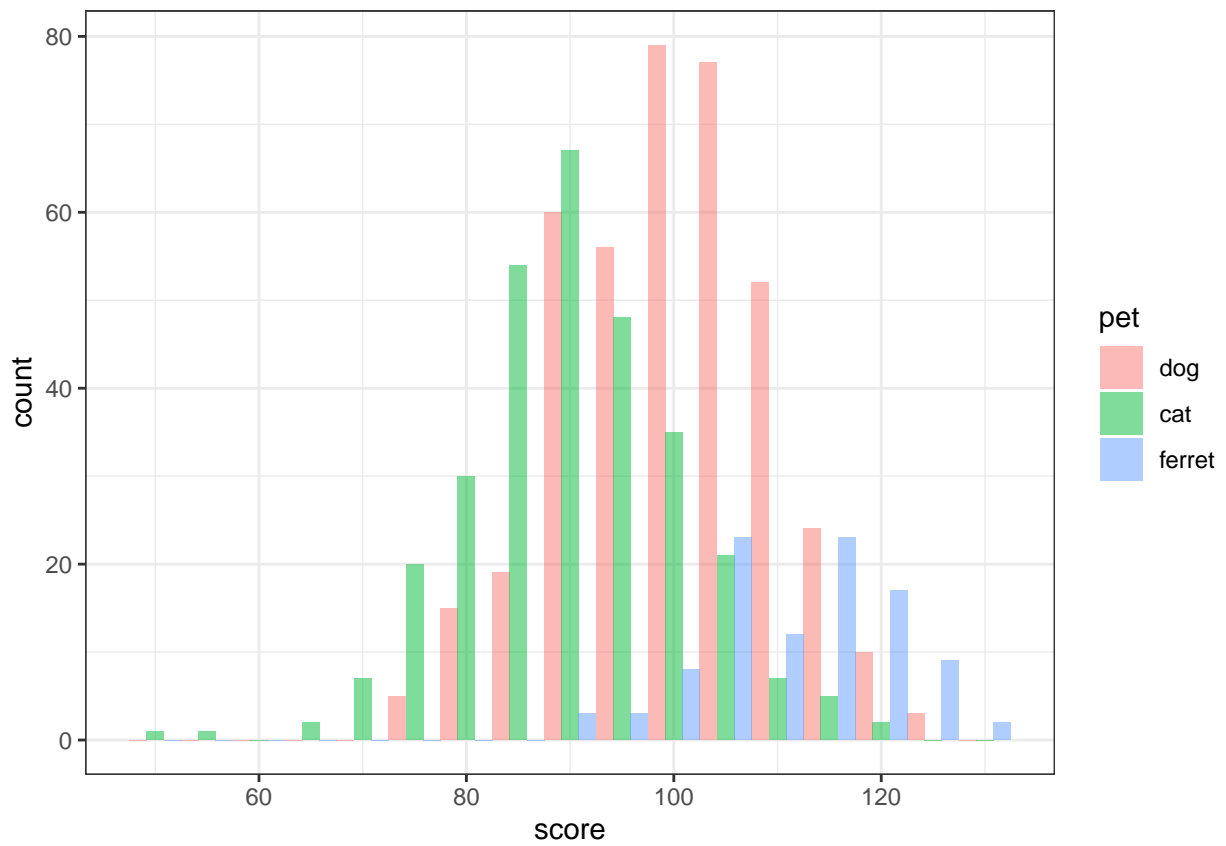
Figure 3.13: Grouped Histogram

```
ggplot(pets, aes(pet, score, fill=pet)) +
  stat_summary(fun = mean, geom = "col", alpha = 0.5) +
  stat_summary(fun.data = mean_se, geom = "errorbar",
               width = 0.25) +
  coord_cartesian(ylim = c(80, 120))
```
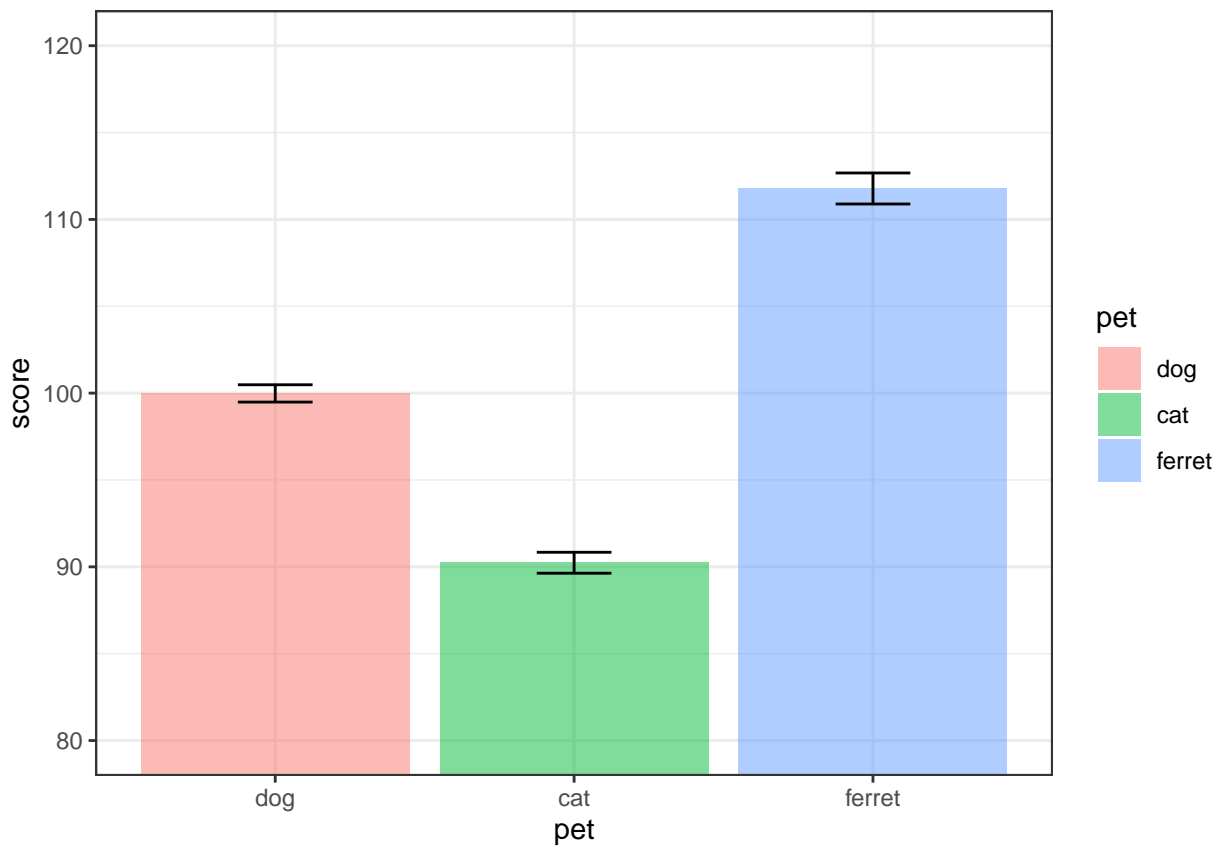

Figure 3.14: Column plot

Try changing the values for `coord_cartesian`. What does this do?

3.6.6 Boxplot

Boxplots are great for representing the distribution of grouped continuous variables. They fix most of the problems with using bar/column plots for continuous data.

```
ggplot(pets, aes(pet, score, fill=pet)) +
  geom_boxplot(alpha = 0.5)
```

3.6.7 Violin plot

Violin pots are like sideways, mirrored density plots. They give even more information than a boxplot about distribution and are especially useful when you have non-normal distributions.
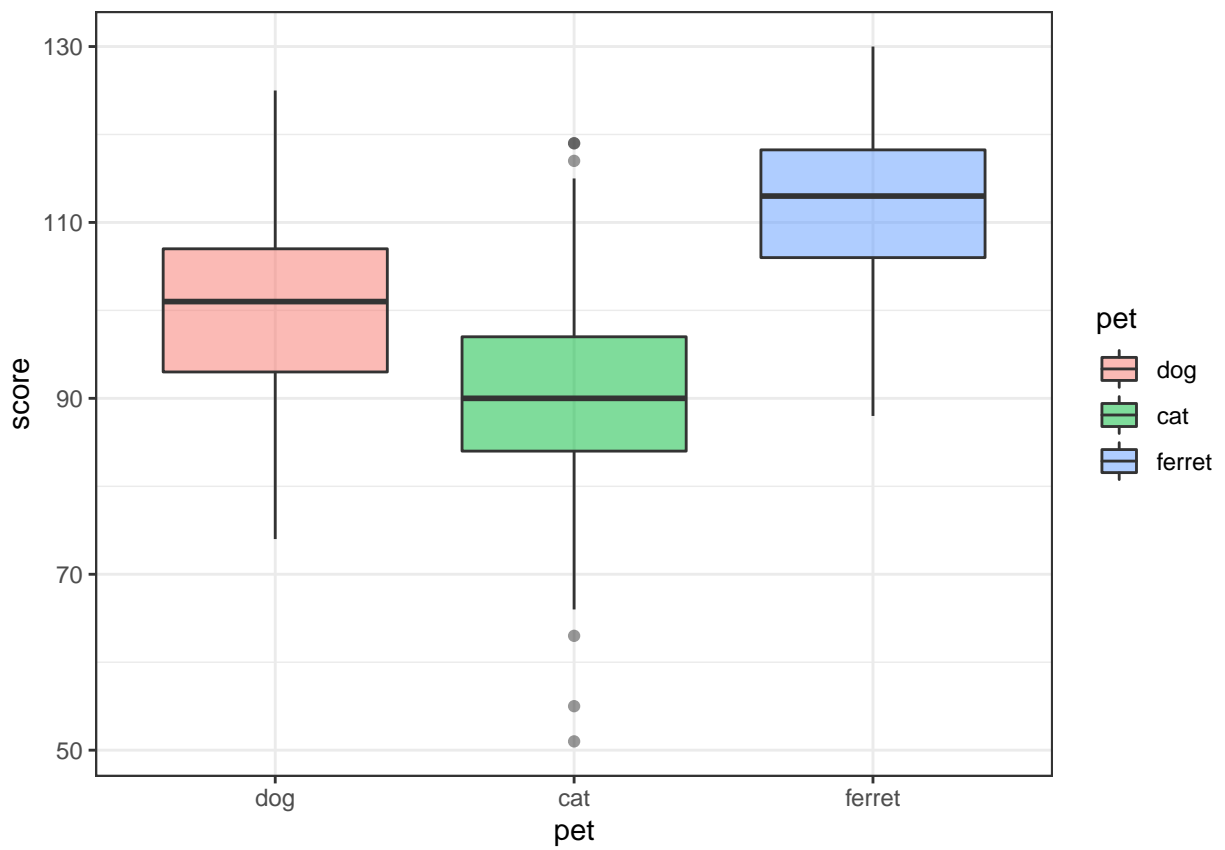
Figure 3.15: Box plot

```
ggplot(pets, aes(pet, score, fill=pet)) +
  geom_violin(draw_quantiles = .5,
              trim = FALSE, alpha = 0.5,)
```
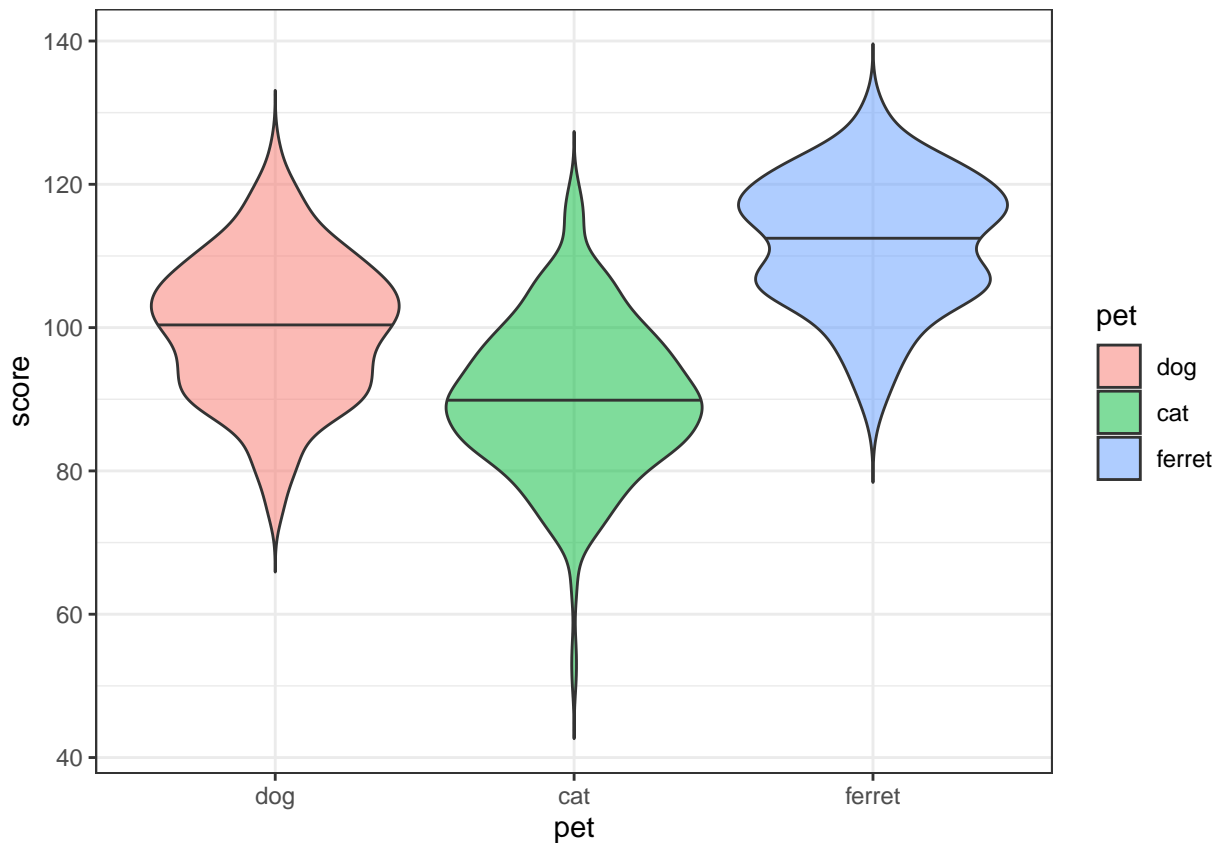

Figure 3.16: Violin plot

> Try changing the `quantile` argument. Set it to a vector of the numbers 0.1 to 0.9 in steps of 0.1.

3.6.8   Vertical intervals

Boxplots and violin plots don't always map well onto inferential stats that use the mean. You can represent the mean and standard error or any other value you can calculate.

Here, we will create a table with the means and standard errors for two groups. We'll learn how to calculate this from raw data in the chapter on data wrangling. We also create a new object called **gg** that sets up the base of the plot.

```
dat <- tibble(
  group = c("A", "B"),
  mean = c(10, 20),
  se = c(2, 3)
)
gg <- ggplot(dat, aes(group, mean,
```