opeRate

Learn to apply R

Adam Rawles

Contents

Chapter 1

opeRate

1.1 Overview

This book is a collection of materials to help users apply their fundamental R knowledge to real programming and analysis. This book is the second in a series of R books I've been working on. The first in the series (teacheR[1]) focuses on the fundamentals of the R language. I would recommend reading teacheR first if you're brand new to the language. It's split into two parts ("For Students" and "For Teachers"). To get the most out of this book, I would suggest that you are at least comfortable with the entirety of the "For Students" section, but it wouldn't hurt to go through the "For Teachers" section while you're at it.

As with the teacheR[2] book, this is a work in progress, so please feel free to make any suggestions or corrections via this book's GitHub repository[3].

1.1.1 Acknowledgements

This book was made possible with the help of those who raised issues and proposed pull requests. With thanks to:

1.2 About Me

I began using R in my second year of university, whilst studying psychology. Like so many others before me, I started using R for a particular project - in my case, it was for an analysis of publication bias - before deciding that I wanted to

[1] https://teacher.arawles.co.uk
[2] teacher.arawles.co.uk
[3] www.github.com/arawles/operate/issues

expand my skillset and learn to apply R to lots of different situations. Because I took this approach however, I didn't really develop a fundamental knowledge of how R worked before I started - I just kind of jumped in at the deep end. As a quick analogy, it was a bit like starting with this book without reading the teacheR book first - I kind of knew what was going on, but I was filling in a lot of gaps along the way.

And so that is why I decided to develop this series of books - to hopefully help anyone who may find themselves in a similar position that I was in those years ago. If you want to use R but feel as though you don't know where to start, then hopefully this book will give you a good overview of some of the different ways that R can be used or applied.

1.3 Using R

In my primary years, analysis in R took me longer than it would take to do the same analysis in something like Excel. And that's okay. R is a complicated and flexible system, and so your first analysis piece will never be particularly efficient. As you stick with it however, and you get used to the methods of automation and a pipeline of execution, you'll find yourself working much more efficiently, performing analyses in half the time. And that's what I hope I can impart with this book; it'll be slow at first, but you'll notice a turning point when you complete your first analysis project in a decent timescale and you'll never look back. Then, before long, you'll have a repertoire of analysis tools at your disposal that make you a crucial member of any data analysis team.

And so in this book we're going to look at some of the common tasks that one might decide to do in R. Keep in mind though that we can't cover everything, so just because it's not in the book doesn't mean that it can't be done!

Chapter 2

tidyverse

As we learnt in the teacheR[1] book, R is supported by thousands of packages that provide extra functionality to the base R experience. One of the most popular sets of packages developed by Hadley Wickham and the RStudio Team is called the tidyverse[2]. The tidyverse is made up of packages designed for data science work that are all underpinned by a common philosophy and a common syntax. At the core of this philosophy is the concept of "tidiness" in data.

The tidyverse is a set of **opinionated** packages. That means that there's (usually) a right way to do things with the package, and there's a wrong way. There's much debate in the R community as to whether relying heavily on opinionated external packages such as those included in the tidyverse is a good thing. Personally, I think that the packages included in the tidyverse are fantastic, and are a large reason why R is thriving today.

So in this book, we're a tidyverse family, and we'll be using the tidyverse packages throughout.

2.1 "Tidy" data

The concept of "tidy" data is something that's important throughout the set of tidyverse packages. There is an in-depth paper[3] published in the Journal of Statistical Software, authored by Hadley Wickham, that describes the concept and requirements of tidy data in full. But for now we're just going to look at the basics.

In order for a dataset to be tidy, it needs to abide by three rules:

[1] teacher.arawles.co.uk
[2] https://www.tidyverse.org/
[3] https://vita.had.co.nz/papers/tidy-data.html

1. Every column is a variable
2. Every row is an observation
3. Every cell is a single value

If a dataset doesn't abide by all three of these rules, then it's a "messy" dataset.

The benefit of having your data in a tidy format is fairly simple; it provides a standard for structuring a dataset. This means that tools can be developed with the assumption that the data will be in a specific format, making the development of these data science tools easier. And this is essentially what underpins the entire tidyverse: get your data in a tidy format and you'll have access to everything the tidyverse provides.

2.2 The pipe %>%

The pipe is a core feature of the tidyverse packages, and has proved so popular in its R implementation that it served as the inspiration for the `|>` function that was introduced into R in version 4.0. For now we're going to look specifically at the tidyverse (or more specifically the `magrittr`) implementation of the pipe but they are very similar.

At the simplest level, the pipe takes the output from whatever is on its left and passes it as the first argument to the expression on the right. Like this:

```
1 %>% print()
```

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

In this example, the 1 is passed as the first argument to the `print()` function, which prints it out.

You can also specify which argument you want the value to be passed as by using the `.` shorthand:

```
1 %>% substr("water", ., 2)
```

```
## [1] "wa"
```

This passes the 1 from the left hand side as the second argument to the `substr()` function (which is the `start` index).

The benefit of the pipe is that it allows you to read code from left to right, rather than from middle outwards. Let's compare two examples with an without the pipe:

```
str <- "water"

print(paste0(substr(str, 1, 1), "ine"))
```

```
## [1] "wine"
```

```
str %>% substr(1,1) %>% paste0("ine") %>% print()
```

```
## [1] "wine"
```

Although the outcome is exactly the same, following the flow of logic is much easier in the bottom example because we can start at the left and more rightwards.

For the purposes of this book that's as much as you need to know. You'll see the pipe used a lot when we're performing multiple data analysis steps, like cleaning then filtering then selecting columns and so on. So as long as you've grasped the simple concept; we're taking the left hand side and passing it to the right, then you'll be fine.

2.3 Quasiquotation

In the teacheR[4] book, we looked at the concept of quasiquotation; selectively quoting and then evaluating parts of an expression. The tidyverse packages rely heavily on this concept, which is why you'll often see (for example) column names passed to the tidyverse packages as variable names rather than character strings, like this:

```
my_df %>%
  dplyr::mutate(new_column = old_column + 1)
```

When you pass a column name like this, the tidyverse packages will quote that input and then evaluate the resulting expression in the context of the current data frame. So essentially, R looks for the `old_column` object in the context of `my_df`. Because dataframes are essentially just lists and lists can be treated like environments, it searches that environment for the `old_column` object and finds it, returning the contents of the column.

[4] https://teacher.arawles.co.uk

2.3.1 tidyselect

Much of this functionality is powered by a package called `{tidyselect}`. The `{tidyselect}` package provides a number of functions for selecting variables from datasets, reducing the time spent laboriously typing out the name of every variable you want to reference.

To better understand the intracicies of the `{tidyselect}` dialect and how it interacts with the tidyverse packages, there are a number of vignettes[5] included in the package that explain things well. For now however, if you can understand the basic concept that variables are evaluated in the specific context of the dataset provided to the function, then you should be fine.

[5] https://tidyselect.r-lib.org/reference/language.html

Chapter 3

Data Analysis

In this chapter, we're going to go through some typical data science tasks and learn how to do them in R. Later on, we'll look at how you should apply these skills in your projects, and some tips for making your project easier to read and share with others.

3.1 Loading

The first step in any data analysis project you'll undertake is getting at least one dataset. Oftentimes, we have less control over the data we use than we would like; receiving odd Excel spreadsheets or text files or files in a proprietary format or whatever. In this chapter, we'll focus on the more typical data formats (rds, CSV and Excel), but we'll also look at how we might extract data from a web API, which is an increasingly common method for data loading. We'll also look briefly at how you can extract data from a SQL database.

3.1.1 RDS

If you need to share an R-specific object (like a linear model created with the `lm()` function) or you're certain that the data never needs to be readable in another program, then you utilise the rds format to save and read data.

To load in .rds files, we use the `readRDS()` function, providing the file path of the file we want to read:

```
my_data <- readRDS(file = "path/to/file")
```

To save an object to an .rds file, you just need to provide the object you want to save and a file path to save it to:

```
saveRDS(my_data, file = "path/to/file")
```

3.1.1.1 Advantages

RDS files are R specific files that contain a serialized version of a specific object. The benefit of R objects is that the object will be preserved in its entirety - you won't lose the column data types when you save it and then load it in again.

3.1.1.2 Disadvantages

RDS files cannot be read natively by other programs. This means that if you're trying to share your dataset and someone wants to open it in, say, Excel, they're going to need to convert it to a different format before they can load it. The RDS format therefore isn't ideal for sharing data outside of the R ecosystem.

3.1.2 CSV

If I have any say in the data format of the files I need to load in, I usually ask for them to be in CSV format. CSV stands for "comma-separated values" and essentially means that the data is stored as one long text string, with each different value or cell separated by a comma (although you will see CSV files with different separators). So for example, a really simple CSV file may look, in its most base format, like this:

```
name,age,
Dave,35,
Simon,60,
Anna,24,
Patricia,75
```

Benefits of the CSV file over something like an Excel file are largely based around simplicity. CSV files are typically smaller and can only have one sheet, meaning that you won't get confused with multiple spreadsheets. Furthermore, values in CSV files are essentially what you see is what you get. With Excel files, sometimes the value that you see in Excel isn't the value that ends up in R (looking at you dates and datetimes). For these reasons, I would suggest using a separated-value file over an Excel file when you can.

3.1.2.1 Loading CSV files

Loading CSV files in R is relatively simple. There are base* functions that come with R to load CSV files but there's also a popular package called `readr` which can be used so I'll cover both.

* They are technically from the `utils` package which comes bundled with R so we'll call it base R.

3.1.2.1.1 Base R To load a CSV file using base R, we'll use the `read.csv()` function:

```
read.csv(file = "path/to/your/file", header = TRUE, ...)
```

The `file` parameters needs the path to your file as a character string. The `header` parameter is used to tell R whether or not your file has column headers.

There are lots of other parameters that can be tweaked for the `read.csv()` function, but we won't go through them here.

3.1.2.1.2 readr The `readr` package comes with a similar function: `read_csv()`. With the exception of a couple of extra parameters in the `read_csv()` function and potentially some better efficiency, there isn't a massive difference between the two.

Using the `read_csv()` function is simple:

```
readr::read_csv(file = "path/to/your/file", col_names = TRUE)
```

In this function, the `header` parameter is replaced with the `col_names` parameter. The `col_names` parameter is very similar, you can say whether your dataset has column headings, or you can provide a character vector of names to be used as column headers.

There are also some extra parameters in the `read_csv()` function that can be useful. The `col_types` parameter lets you specify what datatype each column should be treated as. This can either be provided using the `cols()` helper function like this:

```
readr::read_csv(file = "path/to/file",
                col_names = TRUE,
                col_types = readr::cols(
                  readr::col_character(), readr::col_double()
                ),
                ...
)
```

Or, you can provide a compact string with different letters representing different datatypes:

```
readr::read_csv(file = "path/to/file",
                col_names = TRUE,
                col_types = "cd",
                ...
)
```

The codes for the different datatypes can be found on the documentation page for the `read_csv()` function (type `?read_csv()`).

The `trim_ws` parameter can also be helpful if you have a dataset with lots of trailing whitespace around your values. When set to true, the `read_csv()` function will automatically trim each field before loading it in.

Overall, both functions will give you the same result, so just choose whichever function makes most sense to you and has the parameters you need.

3.1.2.2 Advantages

CSV files can be opened in a number of different software packages, making the CSV format a good candidate for sharing data with people who may not also be using R.

3.1.2.3 Disadvantages

In the interests of simplicity, CSV files don't store information on the **type** of the data in each column, meaning that you need to be careful when you're loading data from a CSV file that the column types you end up with are the ones you want.

The CSV format also isn't fully standardised, meaning that you might come across some files that say they're CSV, but generate errors when they're parsed. It's relatively rare to see a CSV file that is so different that it can't be parsed at all, but it's something worth remembering.

3.1.2.4 Other delimited files

Comma-separated value files are just a form of delimited files that use a comma to separate different values. In the wild you might see files separated with all different kinds of symbols, like pipes (|) or tabs ( ). To load in these types of files, use the `readr::read_delim()` function and specify what's being used to separate the values with the `delim` parameter. `readr::read_csv()` basically just wraps `readr::read_delim()` using `delim = ','` anyway, along as you're comfortable loading in CSV files, you should be well equipped to load in any kind of delimited file.

3.1.3 Excel files

R doesn't have any built-in functions to load Excel files. Instead, you'll need to use a package. One of the more popular packages used to read Excel files is the `readxl` package.

Once you've installed and loaded the `readxl` package. You can use the `read_excel()` function:

```
readxl::read_excel(path = "path/to/file", sheet = NULL, range = NULL, ...)
```

Because Excel files are a little bit more complicated than CSV files, you'll notice that there are some extra parameters. Most notably, the sheet and range parameters can be used to define a subset of the entire Excel file to be loaded. By default, both are set to NULL, which will mean that R will load the entirety of the first sheet.

Like the `readr::read_csv()` function, you can specify column names and types using the `col_names` and `col_types` parameters respectively, and also trim your values using `trim_ws`.

3.1.3.1 Advantages

I have something of a personal vendetta against storing everything in Excel spreadsheets because of the terrible way Excel displays data, so I personally don't think there are too many advantages in using Excel files.

You can have more than one sheet maybe? That's all I've got.

3.1.3.2 Disadvantages

The main disadvantage of Excel files for me is that Excel aggressively formats data for the end user. That is, it's difficult to know what value is actually being stored in a cell based on the value that the end user sees in the cell. Dates are a prime example, Excel will show you the date as a date, but will store it as a number with an origin. That alone isn't a sin at all, but combine that with the fact that Excel has multiple origin dates depending on your Excel version and OS[1], that's a strike in my book.

You could also argue that the xlsx format is a software-specific format, and it kind of is. But because Excel is so ubiquitous now, there are multiple ways of opening and converting xlsx files without ever using Excel, so I don't think that's really a disadvantage.

Overall, if you can send the data in CSV format instead of an Excel file, do that.

[1] https://support.microsoft.com/en-us/office/date-systems-in-excel-e7fe7167-48a9-4b96-bb53-5612a800b487#ID0EBBH=Windows

3.1.4 Web-based APIs

Loading static data from text and Excel files is very common. However, an emerging method of data extraction is via web-based APIs. These web-based APIs allow a user to extract datasets from larger repositories using just an internet connection. This allows for access to larger and more dynamic datasets.

3.1.4.1 What are APIs?

API stands for application programming interface. APIs are essentially just a set of functions for interacting with an application or service. For instance, many of the packages that you'll use will essentially just be forms of API; they provide you with functions to interact with an underlying system or service.

For data extraction, we're going to focus more specifically on web-based APIs. These APIs use the HTTP protocols to accept requests and then return the data requested. Whilst there are multiple *methods* that can be implemented in an API to perform different actions, we're going to focus on `GET` functions. That is, we're purely *getting* something from the API rather than trying to change anything that's stored on the server. You can think of the `GET` method as being read-only.

To start with, we're going to look at exactly how you would interact with an API, but then we'll look at the BMRSr package, which I wrote to make interacting with the Balancing Mechanism and Reporting Service easier.

3.1.4.2 Accessing APIs in R

To access a web-based API in R, we're going to need a connection to the internet, something that can use the HTTP protocol (we're going to use the `httr` package) and potentially some log in credentials for the API. In this case, we're going to just use a test API, but in reality, most APIs require that you use some kind of authentication so that they know who's accessing their data.

As previously mentioned, to extract something from the API, you'll be using the `GET` method. The `httr` package makes this super easy by providing a `GET` function. To this function, we'll need to provide a URL. The `GET` function will then send a GET request to that address and return the response. A really simple GET request could be:

```
httr::GET(url = "http://google.com")
```

```
## Response [http://www.google.com/]
##   Date: 2021-09-07 09:04
##   Status: 200
```

```
##   Content-Type: text/html; charset=ISO-8859-1
##   Size: 12.9 kB
## <!doctype html><html itemscope="" itemtype="http://schema.org/WebPage" lang="...
## var f=this||self;var h,k=[];function l(a){for(var b;a&&(!a.getAttribute||!(b=...
## function n(a,b,c,d,g){var e="";c||-1!==b.search("&ei=")||(e="&ei="+l(d),-1===...
## google.y={};google.sy=[];google.x=function(a,b){if(a)var c=a.id;else{do c=Mat...
## document.documentElement.addEventListener("submit",function(b){var a;if(a=b.t...
## </style><style>body,td,a,p,.h{font-family:arial,sans-serif}body{margin:0;over...
## if (!iesg){document.f&&document.f.q.focus();document.gbqf&&document.gbqf.q.fo...
## }
## })();</script><div id="mngb"><div id=gbar><nobr><b class=gb1>Search</b> <a cl...
## else top.location='/doodles/';};})();</script><input value="ALs-wAMAAAAAYTc5N...
## ...
```

That seems like a really complicated response at first, but when we look at each part, it's quite simple.

- Response
  - This is telling us where we got our response from. In this case, we sent a request to Google, so we got a response from Google.
- Date
  - Fairly self-explanatory - the date and time of the response.
- Content-Type
  - This is telling us what type the response is. In this case, the response is just a HTML page, which is exactly what we expect as that's what you get when you type "google.com" into your browser.
- Size
  - This is the size of the response
- Content
  - Below the size, we see the actual response body. In this case, we've been given the html for the google.com page.

As simple as this example was, it didn't really give us anything interesting back, just the Google homepage. So let's use the GET request to get something more interesting.

We're going to access the jsonplaceholder[2] website, which provides fake APIs for testing. But for now, imagine that this is something like an Instagram database, holding users and their posts and comments.

[2] https://jsonplaceholder.typicode.com/

The first step in accessing an API is to understand that commands the API is expecting. APIs will have what we call **endpoints**. These are paths that we can use to access a certain dataset. For instance, looking at the website, we can see that there are endpoints for lots of different types of data: posts, comments, albums, photos, todos and users. To access an endpoint, we just need to make sure we're using the correct path. So let's try getting a list of users:

```
httr::GET(url = "https://jsonplaceholder.typicode.com/users")
```

```
## Response [https://jsonplaceholder.typicode.com/users]
##   Date: 2021-09-07 09:04
##   Status: 200
##   Content-Type: application/json; charset=utf-8
##   Size: 5.64 kB
## [
##   {
##     "id": 1,
##     "name": "Leanne Graham",
##     "username": "Bret",
##     "email": "Sincere@april.biz",
##     "address": {
##       "street": "Kulas Light",
##       "suite": "Apt. 556",
##       "city": "Gwenborough",
## ...
```

Looking at the content type, we can see that unlike when we sent a request to Google.com, we've got a Content-Type of application/json. JSON is a data structure often used to send data across APIs. We won't go into the structure of it now because R does most of the conversion for us, but if you're interested, there's more info on the JSON structure at www.json.org[3].

Trying to read raw JSON is hard, but `httr` includes functions to help us get it into a better structure for R. Using the `httr::content()` function, `httr` will automatically read the response content and convert it into the format we ask for (via the `as` parameter). For now, we're going to leave the `at` parameter as 'NULL' which guesses the best format for us.

```
response <- httr::GET(url = "https://jsonplaceholder.typicode.com/users")
content <- httr::content(response)
head(content, 1) # we'll just look at the first entry for presentation sake
```

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

[3] https://www.json.org/json-en.html

```
## [[1]]$id
## [1] 1
##
## [[1]]$name
## [1] "Leanne Graham"
##
## [[1]]$username
## [1] "Bret"
##
## [[1]]$email
## [1] "Sincere@april.biz"
##
## [[1]]$address
## [[1]]$address$street
## [1] "Kulas Light"
##
## [[1]]$address$suite
## [1] "Apt. 556"
##
## [[1]]$address$city
## [1] "Gwenborough"
##
## [[1]]$address$zipcode
## [1] "92998-3874"
##
## [[1]]$address$geo
## [[1]]$address$geo$lat
## [1] "-37.3159"
##
## [[1]]$address$geo$lng
## [1] "81.1496"
##
##
##
## [[1]]$phone
## [1] "1-770-736-8031 x56442"
##
## [[1]]$website
## [1] "hildegard.org"
##
## [[1]]$company
## [[1]]$company$name
## [1] "Romaguera-Crona"
##
## [[1]]$company$catchPhrase
## [1] "Multi-layered client-server neural-net"
```

```
##
## [[1]]$company$bs
## [1] "harness real-time e-markets"
```

We can see that R has taken the response and turned it into a list for us. From here, we can then start our analysis.

In many cases however, you won't want a complete list. Instead, you'll want to provide some parameters to limit the data you get back from your endpoint. Most APIs will have a way of doing this. For example, reading the jsonplaceholder website, we can see that we can get all the posts for a specific user by appending the url with "?userId=x". This section of the URL (things after a ?) are called the query part of the URL. So let's try getting all of the posts for the user with ID 1:

```
response <- httr::GET(url = "https://jsonplaceholder.typicode.com/posts?userId=1")
content <- httr::content(response)
head(content, 1) # we'll just look at the first entry for presentation sake
```

```
## [[1]]
## [[1]]$userId
## [1] 1
##
## [[1]]$id
## [1] 1
##
## [[1]]$title
## [1] "sunt aut facere repellat provident occaecati excepturi optio reprehenderit"
##
## [[1]]$body
## [1] "quia et suscipit\nsuscipit recusandae consequuntur expedita et cum\nreprehende
```

Whilst the parameters here are pretty simple, you will come across APIs that accept multiple parameters, making data extraction from an API a very powerful tool.

3.1.4.3 BMRSr

As easy as the above was, interacting with APIs that have several parameters and complicated URLs can get confusing. To this end, many people create packages in R that act as wrappers for various APIs. These packages will then provide you with functions that will automatically create the request, send it and receive and parse the content. You can kind of think about it as an API for an API!

This is what I did for the Balancing Mechanism Reporting Service (BMRS[4]) API. BMRS provides a massive amount of energy-related data, but creating the correct URLs and dealing with the response can be tricky. The BMRSr package that I wrote was designed to help with that.

We'll now go through a quick demo of the BMRSr package. If you're not too bothered about this part, feel free to skip to the next section.

If you're interested, there are a couple of things you'll need:

- The BMRSr package installed
- A free BMRS API key that can be retrieved from the ELEXON portal[5].

Once you've got those two prerequisites, using BMRSr should be quite easy. The main function in the BMRSr package is the `full_request()` function, which will create your URL, send the request, and parse the response depending on your parameters. To do this however, the `full_request()` function needs some parameters:

- `data_item`
  - A data item to retrieve. The BMRS platform holds lots of datasets, and so we need to specify which one we want to retrieve.
- `api_key`
  - Our API_key that we got from the Elexon portal
- parameters
  - Depending on which data_item you chose, you'll need to provide some parameters to filter the data
- `service_type`
  - What format you want the data returned in: values are XML or CSV.

So what parameters do I need? Well, the easiest way to find out is to use the `get_parameters()` function. This will return all of the parameters that can be provided to the `full_request()`.

Let's do an example. Say I want to return data for the B1620 data item, which shows us aggregated generation output per type. So, the first step is to know what parameters I can provide using the `get_parameters()` function:

```
BMRSr::get_parameters("B1620")
```

[4]https://bmreports.com/
[5]https://www.elexonportal.co.uk/

```
## [1] "settlement_date" "period"
```

This tells me that I can provide two parameters in my request - the date and the settlement period. Using this information in my `full_request()` function...

```
bmrs_data <- BMRSr::full_request(data_item = "B1620",
                                 api_key = "put_your_API_key_here",
                                 service_type = "csv",
                                 settlement_date = "01/11/2019",
                                 period = "*") # From reading the API manual,
# I know that this returns all periods
head(bmrs_data, 2)
```

```
## # A tibble: 2 x 13
##   `*Document Type` `Business Type` `Process Type` `Time Series ID` Quantity
##   <chr>            <chr>           <chr>          <chr>                <dbl>
## 1 Actual generati~ Production      Realised       NGET-EMFIP-AGPT~      1636
## 2 Actual generati~ Production      Realised       NGET-EMFIP-AGPT~         0
## # ... with 8 more variables: `Curve Type` <chr>, Resolution <chr>, `Settlement
## #   Date` <date>, `Settlement Period` <dbl>, `Power System Resource
## #   Type` <chr>, `Active Flag` <chr>, `Document ID` <chr>, `Document
## #   RevNum` <dbl>
```

And there we have it, we've retrieved a energy-related dataset from an API using the BMRSr package. There are roughly 101 data items available on BMRS so there's a massive amount of data there for those who want to access it.

3.1.5 Databases

\TO DO

3.2 Cleaning

Loading data is often just the first step in your project. Most of the time, you'll have messy datasets with odd columns and missing data points that you'll need to deal with before you can actually do any meaningful analysis: you'll need to **clean** your data.

Here are some of the more common operations that you'll be doing when it comes to data cleaning:

- Removing/replacing missing values

- Changing column types
- Combining columns
- Renaming columns
- Checking for anomalies

Let's look at how we might do these tasks in R using the `datasets::airquality` dataset as an example.

```
head(datasets::airquality, 10)
```

```
##    Ozone Solar.R Wind Temp Month Day
## 1     41     190  7.4   67     5   1
## 2     36     118  8.0   72     5   2
## 3     12     149 12.6   74     5   3
## 4     18     313 11.5   62     5   4
## 5     NA      NA 14.3   56     5   5
## 6     28      NA 14.9   66     5   6
## 7     23     299  8.6   65     5   7
## 8     19      99 13.8   59     5   8
## 9      8      19 20.1   61     5   9
## 10    NA     194  8.6   69     5  10
```

3.2.1 Missing values

Missing values are common in data science - data collection is often imperfect and so you'll end up with observations or data-points missing. Firstly, you need to decide what you're going to do with those.

The easiest approach to just to remove them, and we can do that with the `dplyr::filter()` function:

```
## To remove rows with NA in one column
datasets::airquality %>%
  dplyr::filter(!is.na(Ozone)) %>%
  head(5)
```

```
##   Ozone Solar.R Wind Temp Month Day
## 1    41     190  7.4   67     5   1
## 2    36     118  8.0   72     5   2
## 3    12     149 12.6   74     5   3
## 4    18     313 11.5   62     5   4
## 5    28      NA 14.9   66     5   6
```

```
## To remove rows with NA in any column
datasets::airquality %>%
  dplyr::filter(dplyr::across(dplyr::everything(), ~!is.na(.x))) %>%
  head(5)
```

```
##   Ozone Solar.R Wind Temp Month Day
## 1    41     190  7.4   67     5   1
## 2    36     118  8.0   72     5   2
## 3    12     149 12.6   74     5   3
## 4    18     313 11.5   62     5   4
## 5    23     299  8.6   65     5   7
```

Another approach to replace them with either the average for that column or with the closest neighbour value (this works better with time series data).

To replace with the nearest value, we can use the `tidyr::fill()` function:

```
## Fill a single column
datasets::airquality %>%
  tidyr::fill(Ozone, .direction = "downup") %>%
  head(5)
```

```
##   Ozone Solar.R Wind Temp Month Day
## 1    41     190  7.4   67     5   1
## 2    36     118  8.0   72     5   2
## 3    12     149 12.6   74     5   3
## 4    18     313 11.5   62     5   4
## 5    18      NA 14.3   56     5   5
```

```
datasets::airquality %>%
  tidyr::fill(dplyr::everything(), .direction = "downup") %>%
  head(5)
```

```
##   Ozone Solar.R Wind Temp Month Day
## 1    41     190  7.4   67     5   1
## 2    36     118  8.0   72     5   2
## 3    12     149 12.6   74     5   3
## 4    18     313 11.5   62     5   4
## 5    18     313 14.3   56     5   5
```

To replace them with the mean, we can either use a package like `zoo`, or we can use the `tidyverse` packages and our own function:

```
replace_with_mean <- function(x) {
  replace(x, is.na(x), mean(x, na.rm = TRUE))
}

datasets::airquality %>%
  dplyr::mutate(dplyr::across(dplyr::everything(), replace_with_mean)) %>%
  head(5)
```

```
##      Ozone  Solar.R Wind Temp Month Day
## 1 41.00000 190.0000  7.4   67     5   1
## 2 36.00000 118.0000  8.0   72     5   2
## 3 12.00000 149.0000 12.6   74     5   3
## 4 18.00000 313.0000 11.5   62     5   4
## 5 42.12931 185.9315 14.3   56     5   5
```

A word of warning here, however. If you're doing complex modelling or very sensitive analyses, filling values like this can be misleading at best. Always think about the best approach for your specific project and what the repercussions of filling empty values might be.

3.2.2 Changing column types

When you import data into R, sometimes the type of the column doesn't match what you want it to be. One way of tackling this is to define your column types when you import the data (as we looked at before), but it's also perfectly acceptable to change the column type after the import.

Probably the most common conversion is from a character string to a date. For this example, we're just going to use some test data:

```
bad_tibble <- tibble::tribble(~bad_date, ~value,
                              "2012/01/01", 100,
                              "2014/06/01", 200)

bad_tibble %>%
  dplyr::mutate(good_date = as.Date(bad_date, format = "%Y/%m/%d"))
```

```
## # A tibble: 2 x 3
##   bad_date   value good_date
##   <chr>      <dbl> <date>
## 1 2012/01/01   100 2012-01-01
## 2 2014/06/01   200 2014-06-01
```

Essentially, all you need to do is wrap your conversion function (e.g. `as.Date()`, `as.character()`) in a `dplyr::mutate()` call and you should be able to change your columns to whatever you need.

3.3 Tidying

Now that you know your dataset is clean, you'll want to get it into a format that is amenable to your analysis. This is the **tidying** stage. To better understand what "tidy" data is and why we're trying to get our data into this format, go back to the tidyverse chapter.

3.3.1 Pivoting columns

There two different forms of data: * Long * Wide

Tidy data takes this long form - as you add more observations, you'll add more rows. Untidy data often takes the wide form - as you add more observations, you'll add more columns.

Let's look at an example. Imagine the following dataset showing the average temperature each year and in each month:

```
head(column_example, 3)
```

```
##   Year Month1 Month2 Month3
## 1 2012     10     20     30
## 2 2013     20     30     40
## 3 2014     30     40     50
```

As you can see, the more months we add, the wider the data is going to get. How would the same data look in the long (tidy) format?

```
head(column_example_long, 3)
```

```
## # A tibble: 3 x 3
##   Year  Month Temperature
##   <chr> <dbl>       <dbl>
## 1 2012      1          10
## 2 2012      2          20
## 3 2012      3          30
```

Now we've got the month as a variable, the more data we add, the longer the dataset is going to get.

Note: One thing to keep in mind is that long data does not always mean tidy, but wide data cann never be tidy. It's a subtle distinction but it's important to remember.

Wide data is very common, usually because it's slightly easier to read or to enter (particularly if you're using Excel), but it's much harder to work with in R. Let's look at how we might convert wide data to tidy-er long data.

3.3.2 Pivoting columns to rows (longer)

Wide data will always be untidy because it breaks the first rule of tidy data: each column should be a separate variable. Let's revisit our wide example:

```
head(column_example, 3)
```

```
##   Year Month1 Month2 Month3
## 1 2012     10     20     30
## 2 2013     20     30     40
## 3 2014     30     40     50
```

In this example dataset, we've clearly broken the first rule of tidy data: each column is not a unique variable. Instead, we've got the same variable (temperature) in different columns for each month.

How can we convert this example dataset to the tidy, long format? Well, we can utilise the concept of pivoting to transform those month columns into two columns; one with the value (the temperature) and the other with the month. In this case, we need to use the `pivot_longer()` function from the `{tidyr}` package, because we want to pivot from the wide format to the long format.

To do this in the simplest way, we just need to tell the function which columns we want to pivot:

```
tidyr::pivot_longer(column_example, cols = c(Month1, Month2, Month3))
```

```
## # A tibble: 9 x 3
##   Year  name   value
##   <chr> <chr>  <dbl>
## 1 2012  Month1    10
## 2 2012  Month2    20
## 3 2012  Month3    30
## 4 2013  Month1    20
```

```
## 5 2013  Month2    30
## 6 2013  Month3    40
## 7 2014  Month1    30
## 8 2014  Month2    40
## 9 2014  Month3    50
```

This is a good start. Now we've converted to long format, we're abiding by the three rules and so we've got a tidy dataset! But there's definitely some improvements to be done. Firstly, "name" and "value" aren't the best names we could come up with for these columns, so we should probably use some new ones. To do this, we just need to provide new names to the `names_to` and `values_to` parameters:

```
tidyr::pivot_longer(column_example,
                    cols = c(Month1, Month2, Month3),
                    names_to = "Month",
                    values_to = "Temperature")
```

```
## # A tibble: 9 x 3
##   Year  Month  Temperature
##   <chr> <chr>        <dbl>
## 1 2012  Month1          10
## 2 2012  Month2          20
## 3 2012  Month3          30
## 4 2013  Month1          20
## 5 2013  Month2          30
## 6 2013  Month3          40
## 7 2014  Month1          30
## 8 2014  Month2          40
## 9 2014  Month3          50
```

Secondly, our Month column has the months as character string and prepended with "Month". Instead, we should store just the month number as a number (or maybe as a factor but we'll go with number). To do that, we can utilise the `names_prefix` parameter to remove matching text from the start of each variable name:

```
tidyr::pivot_longer(column_example,
                    cols = c(Month1, Month2, Month3),
                    names_to = "Month",
                    values_to = "Temperature",
                    names_prefix = "(Month)") # Remove text matching "Month" exactly
```

```
## # A tibble: 9 x 3
```

```
##   Year  Month Temperature
##   <chr> <chr>       <dbl>
## 1 2012  1              10
## 2 2012  2              20
## 3 2012  3              30
## 4 2013  1              20
## 5 2013  2              30
## 6 2013  3              40
## 7 2014  1              30
## 8 2014  2              40
## 9 2014  3              50
```

That's pretty good, but that column is still a character column. We can fix that by using the `names_transform` parameter with a function that we want to use to transform the column:

```
tidyr::pivot_longer(column_example,
                    cols = c(Month1, Month2, Month3),
                    names_to = "Month",
                    values_to = "Temperature",
                    names_transform = list(Month = readr::parse_number))
```

```
## # A tibble: 9 x 3
##   Year  Month Temperature
##   <chr> <dbl>       <dbl>
## 1 2012      1          10
## 2 2012      2          20
## 3 2012      3          30
## 4 2013      1          20
## 5 2013      2          30
## 6 2013      3          40
## 7 2014      1          30
## 8 2014      2          40
## 9 2014      3          50
```

That's better. But what happens now if we get sent the same dataset but with all 12 months worth of data? We'd have to add those extra columns to our `cols` vector. Instead, we can use some `tidyselect` syntax to choose columns based on their features, like how they start:

```
tidy_column_example <- tidyr::pivot_longer(column_example,
                                           cols = tidyselect::starts_with("Month"),
                                           names_to = "Month",
                                           values_to = "Temperature",
```

```
                                        names_transform = list(Month = readr::parse
print(tidy_column_example, n = 3)
```

```
## # A tibble: 9 x 3
##   Year  Month Temperature
##   <chr> <dbl>       <dbl>
## 1 2012      1          10
## 2 2012      2          20
## 3 2012      3          30
## # ... with 6 more rows
```

Now our code will work regardless of how many months there are.

3.3.3 Pivoting rows to columns (wider)

Most of the time you should be going from wide to long, but we'll go through how to do the reverse for the occasions where it's required. To transform the data to the wide format, we use the `pivot_wider()` function. At the simplest level, we just need to provide where the name comes from and where the values come from with the `names_from` and `values_from` parameters respectively:

```
tidyr::pivot_wider(tidy_column_example, names_from = Month, values_from = Temperature)
```

```
## # A tibble: 3 x 4
##   Year    `1`   `2`   `3`
##   <chr> <dbl> <dbl> <dbl>
## 1 2012     10    20    30
## 2 2013     20    30    40
## 3 2014     30    40    50
```

To get our data more like we started, we can use the `names_prefix` argument to prepend the column names with a string:

```
tidyr::pivot_wider(tidy_column_example, names_from = Month, values_from = Temperature,
```

```
## # A tibble: 3 x 4
##   Year  Month1 Month2 Month3
##   <chr>  <dbl>  <dbl>  <dbl>
## 1 2012      10     20     30
## 2 2013      20     30     40
## 3 2014      30     40     50
```

3.4 Grouping

Before we look at adding new columns and summarising our data, we need to understand the concept of grouping our data.

When a data frame has one or more groups, any function that's applied (by `dplyr::mutate()` or `dplyr::summarise()`) is applied for each combination of those groups. So say we have a dataset like this:

```
head(ungrpd, 5)
```

```
##   Year Month Group Value
## 1 2012     1     A    10
## 2 2012     1     B    20
## 3 2012     2     A    30
## 4 2012     2     B    15
## 5 2012     3     A    25
```

If we grouped by Year or by Month, then any function we applied would be applied separately to each value of those groups. So say we wanted get the total values for each Year, we could group by the `Year` column and then just sum the `Value` column. The same logic applies for if we wanted to get the total for each year *and* month; we could group by the `Year` and `Month` and sum the `Value` column and we'd then get a value for each distinct year and month.

To group a dataset, we use the `dplyr::group_by()` function:

```
grpd <- dplyr::group_by(ungrpd, Year)
```

The `dplyr::group_by()` function returns the same dataset but now grouped by the variables you provided. At first, it might not seem as though anything happened - if we print the dataset we still get basically the same output...

```
print(grpd, n = 5)
```

```
## # A tibble: 12 x 4
## # Groups:   Year [2]
##   Year  Month Group Value
##   <chr> <dbl> <chr> <dbl>
## 1 2012      1 A        10
## 2 2012      1 B        20
## 3 2012      2 A        30
## 4 2012      2 B        15
## 5 2012      3 A        25
## # ... with 7 more rows
```

Except now we have a new entry `Groups:`. This tells us that the dataset has been grouped and by what variables. This means that any subsequent summarisation or mutation we do will be done relative those groups.

To test whether a dataset has been grouped, you can use the `dplyr::is_grouped_df()`, and to ungroup a dataset, just use `dplyr::ungroup()`:

```
dplyr::is_grouped_df(grpd)
```

```
## [1] TRUE
```

```
dplyr::is_grouped_df(ungrpd)
```

```
## [1] FALSE
```

```
print(dplyr::ungroup(grpd), n = 5)
```

```
## # A tibble: 12 x 4
##   Year  Month Group Value
##   <chr> <dbl> <chr> <dbl>
## 1 2012      1 A        10
## 2 2012      1 B        20
## 3 2012      2 A        30
## 4 2012      2 B        15
## 5 2012      3 A        25
## # ... with 7 more rows
```

3.5 Mutating

Sometimes you'll want to add new columns to your data. Most of the time, these will be calculated columns that can be created based on one or more of the other columns in the dataset. To create new columns, we use the `dplyr::mutate()` function.

Let's look at how you might mutate your dataset using an example Kaggle dataset[6] that holds information on video game sales:

```
vg_sales <- readr::read_csv("./data/vgsales.csv")
```

```
##
## -- Column specification --------------------------------------------------------
```

[6] https://www.kaggle.com/gregorut/videogamesales

```
## cols(
##   Rank = col_double(),
##   Name = col_character(),
##   Platform = col_character(),
##   Year = col_character(),
##   Genre = col_character(),
##   Publisher = col_character(),
##   NA_Sales = col_double(),
##   EU_Sales = col_double(),
##   JP_Sales = col_double(),
##   Other_Sales = col_double(),
##   Global_Sales = col_double()
## )
```

```
print(vg_sales, n = 5)
```

```
## # A tibble: 16,598 x 11
##    Rank Name  Platform Year  Genre Publisher NA_Sales EU_Sales JP_Sales
##   <dbl> <chr> <chr>    <chr> <chr> <chr>        <dbl>    <dbl>    <dbl>
## 1     1 Wii ~ Wii      2006  Spor~ Nintendo      41.5    29.0      3.77
## 2     2 Supe~ NES      1985  Plat~ Nintendo      29.1     3.58     6.81
## 3     3 Mari~ Wii      2008  Raci~ Nintendo      15.8    12.9      3.79
## 4     4 Wii ~ Wii      2009  Spor~ Nintendo      15.8    11.0      3.28
## 5     5 Poke~ GB       1996  Role~ Nintendo      11.3     8.89    10.2
## # ... with 16,593 more rows, and 2 more variables: Other_Sales <dbl>,
## #   Global_Sales <dbl>
```

First off, let's add something simple; a column that calculates the proportion of EU sales compared to global sales. All we need to do is provide the name of our new column to the `dplyr::mutate()` function and how our new column should be calculated:

```
vg_sales %>%
  dplyr::mutate(EU_Proportion = EU_Sales/Global_Sales) %>%
  dplyr::select(Rank, Name, Platform, Year, Genre, Publisher, EU_Proportion) %>%
  print(n = 5)
```

```
## # A tibble: 16,598 x 7
##    Rank Name                 Platform Year  Genre      Publisher EU_Proportion
##   <dbl> <chr>                <chr>    <chr> <chr>      <chr>             <dbl>
## 1     1 Wii Sports           Wii      2006  Sports     Nintendo         0.351
## 2     2 Super Mario Bros.    NES      1985  Platform   Nintendo         0.0890
## 3     3 Mario Kart Wii       Wii      2008  Racing     Nintendo         0.360
## 4     4 Wii Sports Resort    Wii      2009  Sports     Nintendo         0.334
```

```
## 5     5 Pokemon Red/Pokemon B~ GB       1996  Role-Play~ Nintendo         0.283
## # ... with 16,593 more rows
```

As you can see, we've created a new column called `EU_Proportion` and we've calculated by dividing the number of EU sales by the total number of sales.

While this is certainly a powerful tool, we're not really changing the world here. However, we can create more complicated columns by leveraging the concept of applying our function to groups.

When our dataset is grouped, our `mutate()` call will be evaluated for each permutation of the provided groups. Let's look at an example of creating a cumulative sum for each Publisher:

```
vg_sales %>%
  dplyr::group_by(Publisher) %>%
  dplyr::arrange(as.numeric(Year)) %>% # We need to order from earliest date to latest
  dplyr::mutate(Cumulative_Sales = cumsum(Global_Sales)) %>%
  dplyr::filter(Publisher %in% c("Atari", "Activision")) %>% # Let's just look at Atar
  print(n = 5)
```

```
## # A tibble: 1,338 x 12
## # Groups:   Publisher [2]
##    Rank Name  Platform Year  Genre Publisher NA_Sales EU_Sales JP_Sales
##   <dbl> <chr> <chr>    <chr> <chr> <chr>        <dbl>    <dbl>    <dbl>
## 1   259 Aste~ 2600     1980  Shoo~ Atari         4        0.26        0
## 2   545 Miss~ 2600     1980  Shoo~ Atari         2.56     0.17        0
## 3  1768 Kabo~ 2600     1980  Misc  Activisi~     1.07     0.07        0
## 4  1971 Defe~ 2600     1980  Misc  Atari         0.99     0.05        0
## 5  2671 Boxi~ 2600     1980  Figh~ Activisi~     0.72     0.04        0
## # ... with 1,333 more rows, and 3 more variables: Other_Sales <dbl>,
## #   Global_Sales <dbl>, Cumulative_Sales <dbl>
```

Now we've got a new column showing the total number of sales for each Publisher up to each year. What `{dplyr}` has done is essentially filtered the data down a specific publisher, created a cumulative sum of the `Global_Sales` column, and then done that for each publisher and stitched it back together.

3.6 Summarising

As we saw previously, the base videa game sales dataset has nearly 17,000 entries! That's a lot of data, and we're not really going to able to extract or show any real insights from this data without summarising it to some degree. What we want to do is summarise the data to some degree. We lose some of

the granularity of the data but it allows is to understand the data a bit better and identify patterns.

Summarisation uses exactly the same grouping concept as mutatation does; by grouping the data we change the scope of each application of the function to be limited to that group.

Let's go through an example. Let's say that we want to get the total sales for each publisher in each year. For now, we don't care about genre or platform. To do this, we'll need to group by Publisher and by Year:

```
vg_sales %>%
  dplyr::group_by(Publisher, Year)
```

And then we'll want to sum up the `Global_Sales` column to get our totals:

```
vg_sales %>%
  dplyr::group_by(Publisher, Year) %>%
  dplyr::summarise(Total_Global_Sales = sum(Global_Sales), .groups = "drop")
```

```
## # A tibble: 2,379 x 3
##    Publisher                   Year  Total_Global_Sales
##    <chr>                       <chr>              <dbl>
##  1 10TACLE Studios             2006                0.02
##  2 10TACLE Studios             2007                0.09
##  3 1C Company                  2009                0.01
##  4 1C Company                  2011                0.09
##  5 20th Century Fox Video Games 1981               1.35
##  6 20th Century Fox Video Games 1982               0.59
##  7 2D Boy                      2008                0.04
##  8 3DO                         1998                0.4
##  9 3DO                         1999                4.14
## 10 3DO                         2000                3.08
## # ... with 2,369 more rows
```

```
# The .groups = 'drop' parameter will automatically remove the groups
```

Now we can see that the number of rows has reduced down to just under 2,000, and we've got the total global sales for each of our publishers, and for each year. `{dplyr}` has looked through each publisher and each year that publisher has at least 1 game, and added up the total number of sales, then moved onto the next Publisher/Year combination.

In this case we've used the `sum()` function, but any kind of summary function can be used:

```
vg_sales %>%
  dplyr::group_by(Publisher, Year) %>%
  dplyr::summarise(Total_Global_Sales = sum(Global_Sales),
                   Average_Global_Sales = mean(Global_Sales),
                   .groups = "drop")
```

```
## # A tibble: 2,379 x 4
##    Publisher                     Year  Total_Global_Sales Average_Global_Sales
##    <chr>                         <chr>              <dbl>                <dbl>
##  1 10TACLE Studios               2006                0.02                0.02
##  2 10TACLE Studios               2007                0.09                0.045
##  3 1C Company                    2009                0.01                0.01
##  4 1C Company                    2011                0.09                0.045
##  5 20th Century Fox Video Games 1981                 1.35                0.45
##  6 20th Century Fox Video Games 1982                 0.59                0.295
##  7 2D Boy                        2008                0.04                0.04
##  8 3DO                           1998                0.4                 0.2
##  9 3DO                           1999                4.14                0.69
## 10 3DO                           2000                3.08                0.308
## # ... with 2,369 more rows
```

Because `dplyr::mutate()` and `dplyr::summarise()` both use a similar syntax and both function on grouped datasets, using the correct function when starting out can be tough. The best way to remember which one to use is to ask "How many rows am I expecting back from this?". If the answer is fewer than you've got now, you'll want the `summarise()` function. Otherwise you're looking at `mutate()`.

3.7 Plotting

- `{ggplot2}` & Grammar of Graphics intro
- Basic plots
- Customisation

Chapter 4

Projects

Now we've learnt some basic data science skills, we're going to look at the best way to plan and structure your project. Not all of your analyses will be of sufficient size to warrant a big planning stage, but learning to use a common, separate structure for all of your different projects can really help keep your work clean. This becomes even more important when you begin to combine multiple projects and you want to make sure that they don't slowly start to creep into one. For example, imagine you've previously worked on a project that relied heavily on API data. Then, in your next project, you need to use much of the same data but to a very different end. By utilising this project structure (and more specifically, the idea of "Projects as packages"), you'll be able to easily utilise work from previous projects without duplicating or merging code.

4.1 Workflows

Often the best way to start a data analysis project is to decide what you're workflow should be, breaking your project into distinct stages that are relevant for your particular project.

For instance, if you know that the data you're going to be working on is likely to be littered with mistakes and errors, then you should preemptively allocate a decent amount of your time to the "importing" and "cleaning" steps of your workflow. Similarly, if your end goal is to produce a predictive model at the end, then work in a feedback loop where you inspect and evaluate your model before improving it in the next iteration.

The tidyverse packages we discussed previously are designed to make data science easier, and the workflows we'll look at fit into this philosophy quite nicely.

Each stage of these workflows is usually handled by a different package but with a common syntax and data structure underpinning them all.

4.1.1 Basic Workflow

For now, let's look at a basic workflow and some likely additions or changes you might make depending on your goals.

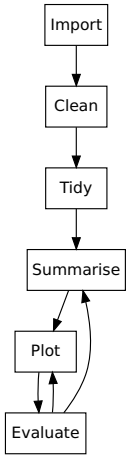
For these basic workflows, we're only going to look at a subset of all of the stages of analysis that you might identify. They are going to be:

- Importing
- Cleaning
- Tidying
- Collating
- Summarising
- Plotting
- Modelling
- Evaluation

4.1.1.1 Example 1: Reporting

In this example, imagine someone has come to you and they want a bit more insight into the data they have. It's currently in a messy spreadsheet, and they want you to load it into R and produce some nice looking graphics to help them understand trends and correlations in their data. They're not bothered about any modelling for now, they just want some pretty graphs.

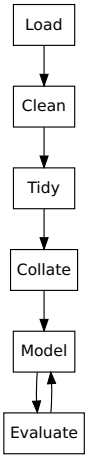
If we mapped out an appropriate workflow for this project, it might look something like this:



We import the data and clean it and tidy it (more on that later), then we summarise it and create our plots. We can then use feedback from our colleague to change how we're summarising or how we're plotting the data to improve our plots until we're happy.

4.1.1.2 Example 2: Modelling

For this example, imagine a colleague has come to you about a modelling project. They have a number of relatively clean datasets that they want to combine and use to produce a predictive model. They're not too worried about plots or graphics, they're more interested in the model itself. If we mapped out a workflow for this project, it might look a bit more like this:



As you can see, we still end with the loop of modelling and then evaluating our model and using the output to update the model, but the steps we take to get there are a bit different.

4.1.1.3 Summary

Hopefully this section has given you an idea of what a typical project workflow could look like. The benefit of splitting your project into these distinct steps is that it can often help you compartmentalise your functions and scripts into distinct stages. We'll look later at structuring your project like an R package, and so splitting your analysis into separate stages can help you construct your project in a portable and replicable way.

4.2 Replicability

The key to a good, robust analysis is replicability. When we say 'replicability', we mean two things:

- That your results can be replicated in other populations and in other settings (i.e. typical 'scientific' replicability)
- That you can easily share your code and method with others who can verify your outputs

If your project is replicable, then it's likely to have fewer issues, make fewer dodgy assumptions, and rely less on the idiosyncracies of your environment or

coding practice. That doesn't mean that everything that you do that can be replicated is immediately correct, but it's a useful credential to have.

For now, we're going to focus on how you can make your *code* more replicable. By that I mean, how you can share your results and your code with others either in your business or instution or even in the open source community in general.

4.2.1 Good practice

There are a few things you can do to make your work immediately more readable for others:

4.2.1.1 Use a consistent naming convention

Take your pick. Everyone has their preference and that's okay. If you like camelCase then go with camelCase. If you like the `_` approach, then go for that. The most important part of your naming convention however is that it's stable. Don't name some of your functions `like_this` and the rest `likeThis`. Not only does it make it harder to read, but every time you do it, a kitten dies. So be consistent.

4.2.1.2 Name your variables as nouns and your functions as verbs

Functions do and variables and objects are. Almost all languages share this distinction with verbs and nouns, so utilise that natural divide to improve how you name your functions and variables. Aim to give your functions names that suggest that they `do` something; and bonus points if you can give it a name that's a verb and also gives a decent description of what the function does. When you name your variables, give them meaningful noun-like names. Say you're doing some climate analysis, a good name for the variable that holds the average rainfall in a year might be `avg_yearly_rainfall` whilst a function that converts a temperature in Celsius to Fahrenheit might be `convert_degrees()`.

4.2.1.3 Use functions where you can

R is a functional language and so using and creating functions is at the very heart of programming in R. Rather than relying on R scripts that need to be run in their entirety to produce your output, by creating functions and using them in your analysis, you can more easily scale your project. For example, imagine your scope is initially to produce a report for the year. Then, after your done, your manager is so impressed with your report, they want you to do the same thing for the last 10 years. If you've used a couple of R scripts and you haven't written any functions, then this is likely going to involve copying and pasting

a whole lot of code and changing the years. Instead, if you use functions to perform your analysis, then creating reports for multiple years can be as simple as changing your dataset.

As a rough rule of thumb, if you find yourself copying and pasting your R code more than twice to do very similar things, you should probably be using a function.

4.2.1.4 State your dependencies

One of the great things about R is the number of packages that are available that let you do all sorts of weird and wonderful things. Unfortunately, because there are so many great packages, it's unlikely that the person you're sharing your code with will have them all installed. If you're sending someone a script, then best practice is to include all the packages you use at the top of your script like this:

```
library(ggplot2)
library(dplyr)
```

An extra step which few people do but can be very helpful is to still prepend your functions with which package they came from like this `dplyr::mutate()`. Not only does this avoid any namespace conflicts where two packages might have functions with the same name and you don't end up using the one you think you are because of the order of your `library()` calls, but it also makes it infinitely easier for anyone reading your code to find the package that a function comes from. Admittedly, this is overkill in a sense because we've already told R which packages we're using with our `library()` calls and R has loaded in the objects from that package, but this practice can really improve the readability of your code.

Later we'll look at designing our project as a package and packages have a different way of stating dependencies for the user, so this is primarily for the case where you're just sending a script or two to someone.

4.2.2 Documentation

When you're working on a project, never think that you'll remember everything when you come back to it. You won't. Instead, imagine you're always writing for code for someone who's never seen it before, because, trust me, when you come back to a project after 6 months you'll have absolutely no idea what you're looking at.

At the heart of writing code that's easy to understand is documentation. Here we'll talk about the two main types of documentation.

4.2.2.1 Function documentation

To get a package on CRAN, all the functions that your package exports needs to be documented. So if you type, say, `?dplyr::mutate()` into the console and hit Enter, you'll be taken to the Help page for the `mutate` function. This ensures that the package users are not left guessing what a parameter is supposed to be, or what they're going to get returned from the function.

Even if you're not ever planning on submitting your work to CRAN, function documentation is extremely useful. The `roxygen` package makes this documentation as easy as possible for package developers, and you can use the same principles in your analysis.

At the very least, you should be documentating the input variables (the `@param` tag in `roxygen`), your return value(s) (the `@return` tag) and maybe an example or two (the `@examples` tag). This will make explaining your code to someone else or relearning it yourself infinitely easier.

4.2.2.2 Long-form documentation

Whilst understanding what each of your functions does is important, it's also important to document how all of the little pieces fit together. To acheive this, it's a good idea to write some long-form documentation, like a README or a vignette.

Long form-documentation is often written in something called RMarkdown. RMarkdown is a spin on markdown[1] that allows you to embed and run R code when generating a document. This can be really useful for explaining your process and sharing your workings.

4.2.2.2.1 READMEs READMEs act as an entrypoint for anyone that stumbles across your package. It should be in the root of your project directory, and should be written in markdown or plain text. Anyone should be able to read your README and understand what the project is doing. I won't go into exactly what should be included because it will depend on the type of project your doing, but you know a good README when you see it.

If you also use a repository system like GitHub, your README will act as the homepage and so should always be present. ## Projects as packages

4.2.2.2.2 Vignettes Vignettes are less defined that READMEs. Vignettes should be an in-depth form of documentation for a concept in your project. For example, say that your project is looking at the level of data quality of various open source APIs. You might have a few different vignettes covering different important concepts in your project:

[1] https://en.wikipedia.org/wiki/Markdown

1. Background
    - This would cover why the how the project came to be, and what the ultimate goal is.
2. Selection of APIs
    - Why were certain APIs chosen and others excluded? That would be explained here.
3. Methodology
    - How are you assessing data quality? Here you would explain your different tests and standards.

Whilst a README serves as a general introduction to the project, vignettes more often serve as in-depth descriptions of certain aspects of the project. Like READMEs, vignettes should be written in markdown (or more likely RMarkdown).

4.3 Projects as Packages

With the last two chapters in mind, to make a good, replicable and easy to understand project, you need a few things:

1. A clear structure
2. Easily replicable code (e.g. consistent functions and clear dependencies)
3. Good documentation (functional and long-form)
4. Maybe a test or two if you want to be confident in your functions and findings

While you could easily fulfill all of these criteria with a standard RStudio project, there's already a type of project that conforms to these standards: *Packages*.

While packages are primarily meant for collating and distributing new functionality, we can utilise the package structure and some of the tools that come along with package development in our projects.

To better understand what a package is and how they are developed and maintained, I would recommend reading Hadley and Jenny Bryan's R Packages book[2] book. Although that book is focused directly on developing packages for submission to CRAN, hopefully you'll be able to identify the parts that are going to be applicable to the way we're using packages.

[2] https://r-pkgs.org/

4.3.1 Structure

4.3.1.1 R code

The structure of R files in a package is pretty simple. Everything goes in the root of the R folder. That means no subfolders. There are also some filenames that should be avoided for normal packages, but we won't worry too much about that for now.

When you then run the `devtools::load_all()` command, `{devtools}` checks that you've got the dependencies listed in the DESCRIPTION file loaded and then loads all of the files in the R folder are then loaded. This means you can make quick changes to your code and then run `devtools::load_all()` to bring those changes into your current environment. This helps prevent working on data or functions that are out of date (which often happens when you're manually sourcing R files).

4.3.1.2 Data

For some projects, you'll want to include a static dataset or datasets that you're basing your analysis on. Packages also have a way of including data in a standard way. Use the `usethis::use_data()` function to bundle an R object with your package. You can make sure that the data needed for your project is included with it.

For other projects, you'll want to use the latest data whenever the analysis is done. For this, I would recommend creating a set of functions to get your data and including those in the package. That will ensure that getting the data needed for your analysis is as simple as possible for people reading your code.

4.3.2 Documentation

4.3.2.1 Functional

To document your functions, I would highly recommend the `roxygen` package. It's by far the easiest way to document your functions. Once you've documentated your functions, you can run `devtools::document()` to automatically generate the documentation that will be seen when someone visits the help page for your function (e.g. via `?your_function`).

4.3.2.2 Long-form

For your long-form documentation, the `usethis::use_readme()` function will provide you with a template for you to build your README.

For vignettes, the `usethis::use_vignette()` function will create the appropriate file and folder for you, so you can just focus on writing.

4.3.3 DESCRIPTION

Every package you download will have a DESCRIPTION file. This file has a number of fields, like who the author is, what license the content is under, and so on. We can utilize many of the fields to help document our project. For example, the Description and Title field are just as relevant for a package-project. Equally, we can use the Version field to keep track of different iterations of our analysis. We can also use this file to state our dependencies.

4.3.3.1 Dependencies

Every package will have an entry called Imports in its DESCRIPTION file. Here, the author is stating every other package that's needed for this package to run. So we can use that same field to state all of the packages that are required for our analysis. That way, when someone installs our package to replicate or check our analysis, all the appropriate dependencies can be installed at the same time.

4.3.4 Abstraction

When I was younger, I wrote a package to create a kind of financial stability report. The report essentially used a number of APIs to pull in macroeconomic data and then create an RMarkdown report displaying the data. Then, a few months later, I started another analysis that used very much the same kind of data but for a different purpose. Now, there were three things I could do:

1. Copy all the code used to get the API data from the original project into the new one
2. Put the original project as a dependency of the new project, importing all of the API data but also all of the other functions
3. Abstract the functions used to get the data from the original project into a new package, and then have both projects use that as dependency.

Given that the title of this section is "Abstraction", what do you think I went with?

Structuring your analysis in this way helps you re-use or repurpose code in the future without having to copy and paste or duplicate any of your previous work.

4.3.5 Example

It can be quite hard to understand the concept of structuring your project as a package without actually giving it a go, so let's go through an example.

4.4 Git