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

Projects

In this section, 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. This idea also links in with some of the theoretical programming concepts we're going to discuss a bit later.

2.1 Workflows

Often the best way to start a data analysis project is to decide what you're workflow should be and roughly how long each bit is going to take.

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 you're 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.

2.1.1 Basic Workflows

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:

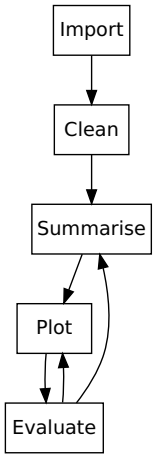
- Importing
- Cleaning*
- Tidying*
- Summarising
- Plotting
- Modelling
- Evaluation

* The difference between data cleaning and data tidying to me is that cleaning refers more to data type conversion, removing NAs and the like. Basically, without cleaning, your analysis isn't going to happen because there's too much noise. When I say 'tidying', that's more getting the data in a format that is amenable to your analysis. You could get by without changing it but it would likely take you much longer or be much less efficient.

2.1.1.1 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:

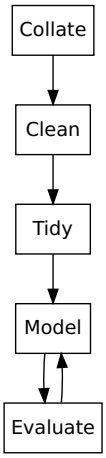


We import the data and clean it, then we summarise it and create our plots. We can then use feedback from our colleague to change how we're summarising or how we're plotting the data to improve our plots until we're happy.

2.1.1.2 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. Our workflow here would be similar to Example 1 but there would be some crucial differences. First, we know that we're going to combine multiple datasets into one which we will then use for our model. This is likely going to involve some data manipulation or 'tidying'. I don't really like using the 'tidying' description because that would suggest that it's the same as 'cleaning' but there's a very useful package for data manipulation and resizing called `tidyr` so we'll stick to 'tidying' for now.

Back to Example 2, 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.

2.1.1.3 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.

2.2 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.

2.2.1 Good practice

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

2.2.1.1 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.

2.2.1.2 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()`.

2.2.1.3 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.

Later on we'll look at the concept of **abstraction** - removing levels of complexity to focus on the core operation - and this tip is heavily related to that concept. For now though, keep this thought in mind: 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.

2.2.1.4 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, 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.

2.2.2 RMarkdown

2.2.3 Shiny

2.3 Projects as packages

Chapter 3

Tidyverse

In this module, we'll take a quick look over some of the packages in the tidyverse[1].

The xaringan presentation for this module can be found here[2].

3.1 tidyr

filler

3.2 dplyr

filler

3.3 stringr

filler

3.4 ggplot2

filler

[1] https://www.tidyverse.org/
[2] /presentations/6_Tidyverse/r_training_tidyverse.html

Chapter 4

Data Analysis

In Chapter 8, we'll look more specifically at how one might do some simple data analysis in R. For a more in-depth view, I would highly recommend Hadley's R4DS[1].

The xaringan presentation for this module can be found here[2].

4.1 Loading data

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 proprietary files or whatever. In this chapter, we'll focus on the more typical data formats (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.

4.1.1 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. So for example, a really simple csv file may look, in its most base format, like this:

```
name,age,
Dave,35,
```

[1] https://r4ds.had.co.nz/

[2] /presentations/3_Data_analysis/r_training_data_analysis_presentation.html

```
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. For these reasons, I would suggest using a separated-value file over an Excel file when you can.

4.1.1.1 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.

4.1.1.1.1 Base 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.

4.1.1.1.2 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.

4.1.2 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`.

4.2 Using 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.

4.2.1 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 URL strings to accept function calls and parameters 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.

4.2.2 Accessing APIs in R

To access a web-based API in R, we're going to need a connection to the internet, 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: 2021-05-21 15:24
##   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,h=[];function k(a){for(var b;a&&(!a.getAttribute||!(b=a.getAttribute("e...
## function m(a,b,c,d,g){var e="";c||-1!=b.search("&ei=")||(e="&ei="+k(d),-1==b....
## 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="AINFCbYAAAAAYKfeo...
## ...
```

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[3] 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.

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: 2021-05-21 15:24
##   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[4].

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.

[3] https://jsonplaceholder.typicode.com/
[4] https://www.json.org/json-en.html

```
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]]
## [[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.

4.2.3 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) 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:

[5] https://www.elexonportal.co.uk/

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

```
## [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.

4.3 Cleaning and tidying data

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.

You'll also need to get it into a format that is amenable to your analysis. This is the **tidying** stage, and because cleaning and tidying data are so tightly related, we'll do these steps at the same time.

4.3.1 Cleaning

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
```

4.3.1.1 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.

4.3.1.2 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.

4.3.1.3 Combining columns

4.4

4.5 Summarisation

4.6 Plotting

Chapter 5

Advanced data analysis

5.1 Plotting

5.2 Modelling

Chapter 6

Shiny

Chapter 7

Theory

When you're first learning R, getting on R and planning little projects and writing code is definitely the best way to learn. Reading to understand *why* you're getting the output that you are or why you're doing something the way you are doing is definitely important, but it's always better to get hands on.

Having said that, one thing that I craved when I was learning R was to understand why people coded the way they did, or why one thing was always recommended over another in StackOverflow answers. I picked it up along the way, but there were many times where I was doing something completely unnecessary or inefficiently because I hadn't been exposed to a discussion about why I shouldn't be doing what I was doing. Similarly, when I eventually did come across an article outlining some of the philosophy or theory underpinning an approach, a little light switch would go and so many more things would click into place.

So this chapter is dedicated purely to some of the simple theory underpinning certain actions in R. This is an *opinionated* piece as I hold a personal opinion on how certain things should be done in R, but that doesn't mean that I'm right. Instead, I hope this section helps you think more deeply about what you're trying to achieve and the best way to get there before you start your next project.

7.1 Abstraction

For me, the biggest change in the quality and efficiency of my code was when I began learning about the concept of abstraction. Abstraction is essentially the process of breaking down complex processes and objects into their base function or quality. From here, we can abstract away levels of complexity, thus creating cleaner looking code.

Abstraction is relevant to every aspect of your projects, from how they are structured to your functions and code and so we're going to spend some time understanding abstraction and how it can be applied.

7.1.1 Definition

Abstraction is the idea of removing levels of complexity. For example, when you press a key on your keyboard and a letter appears on the screen, you don't need to know how the keyboard interfaces with the computer, or how that stroke is eventually turned into coloured pixels on a screen. That degree of complexity has been **abstracted** away.

Another example is a calculator. You type in the numbers and decide what you want to do, and your general goal (say, adding two numbers together) is translated into the practicality of performing that action. Your general goal is translated into lots of little more specific ones.

The idea of abstraction is a very prevalent one in computer science. R itself is an abstraction; it lets you interface with the CPU without having to know everything about it. Understanding abstraction and particularly how it relates to functional programming and R can greatly improve the efficiency of your code. Understanding and applying abstraction is more of an art than a science. By abstracting away complexity, you make things easier for the user but you will take away some of the flexability, and so applying the concept of abstraction to your projects will always be a balancing act.

7.1.2 Abstraction in R

Finding examples of abstraction in existing R code is easy. For instance, in the teacheR[1] book, we looked at how R used method dispatch to find the appropriate method for a particular type of object. If you print a dataframe, for instance, then R will find the appropriate method to print that type of object (a dataframe), and will eventually call the `print.data.frame()` function to do so. But that's not what you have to type in. You just type `print(my_dataframe)` and R takes care of the rest. And that's a good example of how R has abstracted away that complexity, instead focusing on the core goal - printing something.

Applying abstraction to your own R code can greatly improve your code cleanliness and efficiency. One of our main tools for implementing abstraction into our projects are functions. We want to break down our steps into their smallest constituent parts, and aim to create functions that are as simple as possible. We can then use these functions together to solve the overarching issue. The best way to demonstrate how powerful abstraction can be in R is probably through an example. First we'll look at a simple example and then we'll move onto something more complex.

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

7.1.2.1 Example 1 - `plotly` labels

This is a real example of abstraction that I used very recently. The `plotly` package is a great library for producing interactive graphs for Shiny applications and RMarkdown documents. I often use the `ggplotly()` function, which takes a `ggplot` object and converts it to a plotly one.

One of the features of the package is that you can create a tooltip, such that the individual looking at the graph can hover their cursor over the line or bar or whatever and a little box will pop up showing them the value that they're hovering over.

The tooltips are generated automatically from the aesthetics you define in your `ggplot2` call. So if you create a graph with an `x` called `x_val`, then the tooltip will say something like "x_val: 100" when you hover over it. This is fine, but it can look a bit messy when you don't have very nice variable names, which I often don't. Instead, you can create a new aesthetic (like `text`) and provide your values to that aesthetic with nicer names. For example:

```
gplot <- ggplot(data, aes(x = x_val, y = y_val, text = paste0("Better X Label:" = x)))...
plotly_plot <- ggplotly(gplot, tooltip = "text")
```

Now, the tooltip will show "Better X Label: 100" instead of "x: 100". Much better.

But what about when you want to have more than one variable in the label? You could try adding more aesthetics and including them to the `tooltip` parameter (`tooltip = c("text", "label", etc.)`), but that could be tough if you've got more than a couple of aesthetics.

So let's think less about the here and now and try and break this into the simplest possible terms. We want a function that will need to accept a new name for our variable, the variable values, and it will need to be able to accept any number of variables. After a bit of trial and error, this is the function I created:

```
plotly_label <- function(labels) {
  raw_lists <- purrr::map2(names(labels), labels, function(name, values) {
    purrr::map(values, ~paste0(name, ": ", .x))
  })
  purrr::pmap(raw_lists, function(...) paste(..., sep = "<br>"))
}
```

This function accepts a list of name-value pairs, with the name representing the new name of the variable. This function then returns new labels (separated by a break) that can be used by `plotly`. So for instance, if I wanted to have labels for both my `x` and `y` variables, I could just do this:

```
gplot <- ggplot(data, aes(x = x_val, y = y_val, text = plotly_label(list("Better X Labe
plotly_plot <- ggplotly(gplot, tooltip = "text")
```

We haven't changed the world here, but we have created a function that can be used in multiple situations. We haven't hard-coded anything in, meaning that theoretically we could create a tooltip with thousands of variables.

7.1.2.2 Example 2 - Plotting

For our first example, let's imagine you want to create a full suite of graphics for a reporting project you're working on. You're going to want to create 10 different line charts and 5 different bar charts using different variables from 3 different datasets. Let's look at some different ways we could go about doing this, applying different levels of abstraction:

7.1.2.2.1 Approach 1 You could just write out the code needed for each plot. For example:

```
library(ggplot2)
linechart1 <- ggplot(dataset1, aes(x = x, y = y, colour = groups)) +
  geom_line()
linechart2 <- ggplot(dataset2, aes(x = x, y = y, colour = second_grouping)) +
  geom_line()
.
.
.
```

After writing our all our code, we'd have 15 different graphs, but a lot of code and a lot of repitition. There's lots of code here that's being shared and every time the same call or section is duplicated, that's twice the code that we might potentially have to debug.

7.1.2.2.2 Approach 2 A different approach may be to create a different function for each plot. The function would take the dataset and spit out the graphic, a bit like this:

```
library(ggplot2)
create_first_linechart <- function(data) {
  ggplot(data, aes(x = column1, y = column2, colour = column3)) +
    geom_line()
}
.
.
.
```

We would eventually have 15 different functions - one for each plot - that we could call to produce all of our graphics. Like this:

```
create_first_linechart(dataset1)
create_second_linechart(dataset2)
create_third_linechart(dataset3)
.
.
```

This is a bit cleaner than our first approach because we've separated out our graph-generation logic and it's certainly a step in the right direction, but there's still just as much repetition. Similarly, if there's an error, we're still going to have to debug each function separately.

7.1.2.2.3 Approach 3 Building on the use of functions, we could create a function that will create all of the line charts and then a separate function that will create all of our bar charts. Our function to create our line charts might look like this:

```
create_linechart <- function(data, x, y, colour) {
  ggplot(data, aes(x = {{ x }}, y = {{ y }}, colour = {{ colour }})) +
    geom_line()
}
create_barchart <- function(data, x, y, colour) {
  ggplot(data, aes(x = {{ x }}, y = {{ y }}, fill = {{ colour }})) +
    geom_col()
}
```

The `{{ }}` brackets let us pass column names from our function to the `aes()` function in `ggplot2`.

Then we'd still have to make 15 separate calls to create our graphics, but we'd have much less code to debug if something went wrong because we're only relying on two functions, not the 15 we were before. Similarly, if we wanted to change something, we'd just have to change the two functions we created and that change would be propogated to all the 15 graphics.

7.1.2.3 Conclusion

Ultimately, we've utilised the concept of abstraction to simplify our code and make debugging easier. The goal when writing code isn't to write it perfectly first time round - in fact, aiming for perfection off the bat can often be detrimental to your work in the long run - but to strike a good balance of simplicity and functionality. There isn't often a 'perfect' amount of abstraction - it just

doesn't really work that way, but the more you program and write code in R and the more you think about your problem and what the core goal you're trying to acheive is, the better your code will become.

7.2 Returns

As R is a functional language, functions (and therefore the values that they return) are an important thing to understand as a user. As we learned in the teacheR[2] book, when you write a function, the return value will be whatever was last evaluated in that function definition, unless you specified a `return()` call. This "early return" strategy, however, is often a point of contention. Let's have a look at the different points of view:

1. **There shouldn't be any early returns**

At one extreme, there are people who teach that you shouldn't return something early from a function (unless there's an error). In practice, this might look something like this:

```
late_bird <- function(x, y, w, add_w = TRUE) {
  ret <- x + y
  if (add_w) {
    ret <- ret + w
  }
  ret
}
```

So what's going on here? Well, we're creating a variable (called `ret`) and always returning that at the end of our function. The value of `ret` changes depending on our parameters, but we always return the value of that `ret` variable.

On the one hand, this makes it clear *which* variable is being returned. But on the other, it's not always clear *what* the value of that variable is. We have to scan down the whole body of the function to see what happens to ret, even though if we set `add_w` to FALSE, nothing happens after the original call.

2. **Never use a common return variable**

At the other end, we could completely avoid return placeholder variables:

[2] http://arawles.co.uk/teacher

```
early_bird <- function(x, y, w, add_w = TRUE) {
  if (add_w) {
    return(x + y + w)
  } else {
    return(x + y)
  }
}
```

Unlike the first example, we only have to read down until the path that we've chosen is finished. For example, if `add_w` is TRUE, then we only need to read down until the first `return()` call and we know what we're going to get. However, this approach would probably be more complicated if we had more than 2 paths or if we're doing complicated actions. Plus, we're duplicating our code here a bit by specifying the `x + y` part in both `return()` calls.

3. **Return early when possible**

And finally, we reach a more middle-of-the-road approach:

```
middle_bird <- function(x, y, w, add_w = TRUE) {
  ret <- x + y
  if (add_w) {
    return(ret + w)
  }
  ret
}
```

Here we use an intermediate variable to avoid duplicating our `x + y` operation, but then we return when we're ready, meaning that someone doesn't have to read to the bottom of the function if they've chosen the `add_w` path.

And herein lies the crux of the issue. Early returns have been a hot topic since the inception of computer programming, and people will continue to have their opinions on what the correct approach should be, so here's mine:

The final return value should follow the "happy" path. That is, if you used the function with its default parameters, it should reach the end and return the final evaluation. Otherwise, you should probably be returning early. This way, when people first glance at the function, they can easily understand the logic and the "default" return value. From there, they can then monitor the edge cases to understand how the return value changes.

So what does this look like exactly? Well, Example 3 is close, but it doesn't return via the final evaluation for the "happy" path. Let's fix that:

```
adams_bird <- function(x, y, w, add_w = TRUE) {
  ret <- x + y
  if (!add_w) {
    return(ret)
  }
  ret + w

}
```

Now, the "happy" path works its way all the way down to the final expression `ret + w`. But when we change the default value for `add_w` and then deviate from the strict "happy" path, we see an early return call when we're ready.

Of course, this approach won't always be the easiest to understand - for example, if you have a parameter that changes the path at multiple points, then you won't be able to return until later in the function anyway, but to me this approach is the most conducive to readable code.