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 ideally meant as a follow on from my introductory book teacheR, but anyone with a simple understanding of the fundamentals of R such as the basic data structures and functions should be able to benefit.

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

1.2 About Me

I've been using R for roughly 4 years now, and it's essentially the language I do the vast majority of my work in. I say the vast majority, because data analysis work usually requires some crossover work with things like SQL servers or APIs or similar, but we'll cover those in time.

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.

Chapter 2

Tidyverse

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

The xaringan presentation for this module can be found here.

2.1 tidyr

filler

2.2 dplyr

filler

2.3 stringr

filler

2.4 ggplot2

filler

Chapter 3

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.

The xaringan presentation for this module can be found here.

3.1 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.

3.1.1 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,
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.

3.1.1.1 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.

3.1.1.1.1 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.

3.1.1.1.2 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.

3.1.2 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`.

3.2 APIs (Advanced)

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.

3.2.1 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.

3.2.2 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: 2020-11-06 14:55
##   Status: 200
##   Content-Type: text/html; charset=ISO-8859-1
##   Size: 13 kB
## <!doctype html><html itemscope="" itemtype="http://schema.org/WebPage" lang="...
## document.documentElement.addEventListener("submit",function(b){var a;if(a=b.t...
## var a=window.location,b=a.href.indexOf("#");if(0<=b){var c=a.href.substring(b...
## </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="AINFCbYAAAAAX6Vx6...
## var c=this||self,e=/^[\w+/_-]+[=]{0,2}$/,f=null,g=function(a){return(a=a.quer...
## a!=c?a=g(a.document):(null===f&&(f=g(c.document)),a=f);a&&b.setAttribute("non...
## ...
```

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 actually 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 [https://jsonplaceholder.typicode.com/] 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. Looking at the website, we can see that we can access lots of different types of data: posts, comments, albums, photos, todos and users. We can also see that to list each one, we just need to add /whatever to the end of our URL. So let's try getting a list of users:

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

```
## Response [https://jsonplaceholder.typicode.com/users]
##   Date: 2020-11-06 14:55
##   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 here.

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. For now, we're going to leave it at '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]]
## [[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. 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". 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\nreprehender
```

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.

3.2.3 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 here.

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 B1610 data item, which shows us generation output per generation unit. So, the first step is to know what parameters I can provide using the `get_parameters()` function:

```
BMRSr::get_parameters("B1610")
```

```
## [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...

```
BMRSr::full_request(data_item = "B1610",
                    api_key = "put_your_API_key_here",
                    service_type = "csv",
                    settlement_date = as.Date("2019/06/01"),
                    period = "*") # From reading the API manual,
                                  # I know that this returns all periods
```

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.

3.3 Cleaning data

filler

3.4 Summary statistics

filler

3.5 Plots

Chapter 4

Programming in R

In this chapter, we'll look at some of the programming concepts and syntax in R.

The xaringan presentation for this module can be found here.

4.1 User-defined functions

filler

4.2 For loops

filler

4.3 If/else statements

filler

4.4 Applying functions

filler

Chapter 5

Advanced data analysis

Chapter 6

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 unneccessary 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 *opionated* 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.

6.1 Abstraction

6.1.1 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.

6.1.2 Abstraction in R

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. And because R is a functional programming language, abstraction is an important thing to understand to write the best possible code.

6.2 Organisation

TO DO

6.2.1 Packages as projects

TO DO

6.3 Iteration

TO DO

6.3.1 For loops and lists

TO DO