Wrangle data in JSON files

library(tidyverse, warn.conflicts = FALSE)

## ── Attaching core tidyverse packages ──────────────────────── tidyverse 2.0.0 ──  
## ✔ dplyr 1.1.4 ✔ readr 2.1.5  
## ✔ forcats 1.0.0 ✔ stringr 1.5.1  
## ✔ ggplot2 3.5.1 ✔ tibble 3.2.1  
## ✔ lubridate 1.9.3 ✔ tidyr 1.3.1  
## ✔ purrr 1.0.2   
## ── Conflicts ────────────────────────────────────────── tidyverse\_conflicts() ──  
## ✖ dplyr::filter() masks stats::filter()  
## ✖ dplyr::lag() masks stats::lag()  
## ℹ Use the conflicted package (<http://conflicted.r-lib.org/>) to force all conflicts to become errors

library(jsonlite, warn.conflicts = FALSE)  
library(RCurl)

##   
## Attaching package: 'RCurl'  
##   
## The following object is masked from 'package:tidyr':  
##   
## complete

## What is JSON

**J**ava **S**cript **O**bject **N**otation

A data format originally used by web developers (JavaScript) has grown immensely popular for capturing structured data. Very often you get such data from web-based services, like here:

<https://data.police.uk/>

Let’s explore their API service [(example in their documentation)](https://data.police.uk/docs/method/crime-street/): <https://data.police.uk/api/crimes-street/all-crime?date=2024-01&lat=52.629729&lng=-1.131592>

An API usually has a documentation where they tell you how to modify the URL to require different pieces of information. but you need the URL of the API and then you add their commands to it. You introduce the first with a question mark and all others with an ampersand like above. The API URL here is https://data.police.uk/api/crimes-street/all-crime and the commands are

* date=2024-01
* lat=52.629729 (GPS latitude)
* lng=-1.131592 (GPS longitude)

So this request is clearly for all crimes in a place determined by GPS coordinates in January 2024.

We will not examine the UK police API in detail to find out what else we could add and just take this data set.

View this entire API request URL in the browser to see the structure you get. Look at the tabs: JSON, Raw data, Headers.

Here is a reference to the JSON format: <https://www.w3schools.com/js/js_json_syntax.asp>. In JavaScript, when you read a JSON file, it becomes a JavaScript object. When we meet JSON outside a JavaScript, it is **JSON strings** we work with.

This is one of many ways to request a response from the API programmatically, without manual download from the browser:

crimes\_json <- RCurl::httpGET("https://data.police.uk/api/crimes-street/all-crime?date=2024-01&lat=52.629729&lng=-1.131592")

What is it that you get?

crimes\_json %>% str()

## chr "[{\"category\":\"anti-social-behaviour\",\"location\_type\":\"Force\",\"location\":{\"latitude\":\"52.637041\",\"| \_\_truncated\_\_  
## - attr(\*, "Content-Type")= Named chr "application/json"  
## ..- attr(\*, "names")= chr ""

length(crimes\_json)

## [1] 1

If you read this in Java Script, you could immediately see a structure, but here it is just a string. However, R can make sense of the structure with dedicated libraries. We will use the jsonlite library.

jsonlite translates the JSON elements to their own data structures, so you can handle it like a list of vectors, data frames, and lists.

This would be a JSON object containing three properties with values (examples from the W3C JSON tutorial). We parse it with R and it becomes a list.

fromJSON('{"name":"John", "age":31, "city":"New York"}')

## $name  
## [1] "John"  
##   
## $age  
## [1] 31  
##   
## $city  
## [1] "New York"

List is a data structure in base R, created and accessed like this:

mylist <- list("first\_element" = c("a", "b", "c"),   
 "second\_element" = tibble("name" = c("John", "Mary", "Dale"),   
 "age" = c("3", "10", "9")  
 ))  
str(mylist)

## List of 2  
## $ first\_element : chr [1:3] "a" "b" "c"  
## $ second\_element: tibble [3 × 2] (S3: tbl\_df/tbl/data.frame)  
## ..$ name: chr [1:3] "John" "Mary" "Dale"  
## ..$ age : chr [1:3] "3" "10" "9"

mylist$second\_element # gets you the second element, which is a data frame (tibble)

## # A tibble: 3 × 2  
## name age   
## <chr> <chr>  
## 1 John 3   
## 2 Mary 10   
## 3 Dale 9

mylist[[2]] # also gets you directly the second element

## # A tibble: 3 × 2  
## name age   
## <chr> <chr>  
## 1 John 3   
## 2 Mary 10   
## 3 Dale 9

mylist[2] # !!! is a list with the only element "second\_element"

## $second\_element  
## # A tibble: 3 × 2  
## name age   
## <chr> <chr>  
## 1 John 3   
## 2 Mary 10   
## 3 Dale 9

str(mylist[2])

## List of 1  
## $ second\_element: tibble [3 × 2] (S3: tbl\_df/tbl/data.frame)  
## ..$ name: chr [1:3] "John" "Mary" "Dale"  
## ..$ age : chr [1:3] "3" "10" "9"

This is a JSON element consisting of an array of property-value pairs.

fromJSON('{"employees":[  
 { "firstName":"John", "lastName":"Doe" },  
 { "firstName":"Anna", "lastName":"Smith" },  
 { "firstName":"Peter", "lastName":"Jones" }  
]}', simplifyDataFrame = TRUE) #default: creates data frames from lists if possible

## $employees  
## firstName lastName  
## 1 John Doe  
## 2 Anna Smith  
## 3 Peter Jones

#loses the info that the element is called employees

When it translates to a data frame, can you see the logic? What is in rows, columns?

Sometimes you may not want to get data frames right away, here is how you forbid it. fromJSON always tries to simplify. If you wanted a maximally true translation, you would have to save the JSON string to a file and call a different function: read\_json() (it will not be demonstrated here).

fromJSON('{"employees":[  
 { "firstName":"John", "lastName":"Doe" },  
 { "firstName":"Anna", "lastName":"Smith" },  
 { "firstName":"Peter", "lastName":"Jones" }  
]}', simplifyDataFrame = FALSE) # overrides the simplifying to data frames

## $employees  
## $employees[[1]]  
## $employees[[1]]$firstName  
## [1] "John"  
##   
## $employees[[1]]$lastName  
## [1] "Doe"  
##   
##   
## $employees[[2]]  
## $employees[[2]]$firstName  
## [1] "Anna"  
##   
## $employees[[2]]$lastName  
## [1] "Smith"  
##   
##   
## $employees[[3]]  
## $employees[[3]]$firstName  
## [1] "Peter"  
##   
## $employees[[3]]$lastName  
## [1] "Jones"

Now back to the UK police API and street crime data. we had this below, and it was a string.

crimes\_json <- httpGET("https://data.police.uk/api/crimes-street/all-crime?date=2024-01&lat=52.629729&lng=-1.131592")   
str(crimes\_json)

## chr "[{\"category\":\"anti-social-behaviour\",\"location\_type\":\"Force\",\"location\":{\"latitude\":\"52.637041\",\"| \_\_truncated\_\_  
## - attr(\*, "Content-Type")= Named chr "application/json"  
## ..- attr(\*, "names")= chr ""

But you know that JSON means structured data! Use jsonlite::fromJSON() to *parse* (interpret) it in R.

crimes <- fromJSON(crimes\_json)

It displays as a data frame in the Environment pane, but that is just the view.

The truth is shown here:

str(crimes)

## 'data.frame': 1472 obs. of 9 variables:  
## $ category : chr "anti-social-behaviour" "anti-social-behaviour" "anti-social-behaviour" "anti-social-behaviour" ...  
## $ location\_type : chr "Force" "Force" "Force" "Force" ...  
## $ location :'data.frame': 1472 obs. of 3 variables:  
## ..$ latitude : chr "52.637041" "52.632052" "52.634503" "52.635078" ...  
## ..$ street :'data.frame': 1472 obs. of 2 variables:  
## .. ..$ id : int 1738379 1737636 1737385 1737968 1737986 1738829 1738882 1738541 1739337 1737598 ...  
## .. ..$ name: chr "On or near Parking Area" "On or near Supermarket" "On or near Kate Street" "On or near St Nicholas Place" ...  
## ..$ longitude: chr "-1.131222" "-1.146096" "-1.149403" "-1.139462" ...  
## $ context : chr "" "" "" "" ...  
## $ outcome\_status :'data.frame': 1472 obs. of 2 variables:  
## ..$ category: chr NA NA NA NA ...  
## ..$ date : chr NA NA NA NA ...  
## $ persistent\_id : chr "" "" "" "" ...  
## $ id : int 116209089 116208620 116208619 116208618 116208873 116208844 116208610 116209040 116208916 116209044 ...  
## $ location\_subtype: chr "" "" "" "" ...  
## $ month : chr "2024-01" "2024-01" "2024-01" "2024-01" ...

and here:

crimes %>% slice\_head(n = 5)

## category location\_type location.latitude location.street.id  
## 1 anti-social-behaviour Force 52.637041 1738379  
## 2 anti-social-behaviour Force 52.632052 1737636  
## 3 anti-social-behaviour Force 52.634503 1737385  
## 4 anti-social-behaviour Force 52.635078 1737968  
## 5 anti-social-behaviour Force 52.627432 1737986  
## location.street.name location.longitude context  
## 1 On or near Parking Area -1.131222   
## 2 On or near Supermarket -1.146096   
## 3 On or near Kate Street -1.149403   
## 4 On or near St Nicholas Place -1.139462   
## 5 On or near Jarrom Street -1.138755   
## outcome\_status.category outcome\_status.date persistent\_id id  
## 1 <NA> <NA> 116209089  
## 2 <NA> <NA> 116208620  
## 3 <NA> <NA> 116208619  
## 4 <NA> <NA> 116208618  
## 5 <NA> <NA> 116208873  
## location\_subtype month  
## 1 2024-01  
## 2 2024-01  
## 3 2024-01  
## 4 2024-01  
## 5 2024-01

and here:

colnames(crimes)

## [1] "category" "location\_type" "location" "context"   
## [5] "outcome\_status" "persistent\_id" "id" "location\_subtype"  
## [9] "month"

Nevertheless, when you know the structure, you can sometimes delve in with dplyr without further wrangling:

crimes %>% group\_by(location$street$name) %>% count()

## # A tibble: 328 × 2  
## # Groups: location$street$name [328]  
## `location$street$name` n  
## <chr> <int>  
## 1 Leicester (station) 9  
## 2 On or near A594 1  
## 3 On or near Abbey Street 9  
## 4 On or near Abingdon Road 6  
## 5 On or near Adderley Road 2  
## 6 On or near Albion Street 3  
## 7 On or near All Saints Road 6  
## 8 On or near Andover Street 9  
## 9 On or near Andrewes Close 1  
## 10 On or near Andrewes Street 1  
## # ℹ 318 more rows

But you may want to make this a proper data frame with columns being just vectors, not lists! It is called **rectangling** and you do this well with the tidyr library. Here is a wonderful overview of the options you have: <https://tidyr.tidyverse.org/articles/rectangle.html>

To make this data set a flat data frame, you need to extract the nested columns to the surface. First examine the location column.

str(crimes$location)

## 'data.frame': 1472 obs. of 3 variables:  
## $ latitude : chr "52.637041" "52.632052" "52.634503" "52.635078" ...  
## $ street :'data.frame': 1472 obs. of 2 variables:  
## ..$ id : int 1738379 1737636 1737385 1737968 1737986 1738829 1738882 1738541 1739337 1737598 ...  
## ..$ name: chr "On or near Parking Area" "On or near Supermarket" "On or near Kate Street" "On or near St Nicholas Place" ...  
## $ longitude: chr "-1.131222" "-1.146096" "-1.149403" "-1.139462" ...

So, there are two vector columns, latitude and longitude, and one nested column street throughout all rows. That’s easy to unpack:

crimes <- crimes %>% unnest\_wider(col = location)  
crimes %>% slice\_head(n = 3)

## # A tibble: 3 × 11  
## category location\_type latitude street$id longitude context  
## <chr> <chr> <chr> <int> <chr> <chr>   
## 1 anti-social-behaviour Force 52.637041 1738379 -1.131222 ""   
## 2 anti-social-behaviour Force 52.632052 1737636 -1.146096 ""   
## 3 anti-social-behaviour Force 52.634503 1737385 -1.149403 ""   
## # ℹ 6 more variables: street$name <chr>, outcome\_status <df[,2]>,  
## # persistent\_id <chr>, id <int>, location\_subtype <chr>, month <chr>

Now let’s examine the street column, just in case:

crimes$street %>% str()

## 'data.frame': 1472 obs. of 2 variables:  
## $ id : int 1738379 1737636 1737385 1737968 1737986 1738829 1738882 1738541 1739337 1737598 ...  
## $ name: chr "On or near Parking Area" "On or near Supermarket" "On or near Kate Street" "On or near St Nicholas Place" ...

This also looks as if there were one id and one street name per row, so should be easily unnested the same way:

crimes %>% unnest\_wider(col = street)

## Error in `unnest\_wider()`:  
## ! Can't duplicate names between the affected columns and the original  
## data.  
## ✖ These names are duplicated:  
## ℹ `id`, from `street`.  
## ℹ Use `names\_sep` to disambiguate using the column name.  
## ℹ Or use `names\_repair` to specify a repair strategy.

The error occurs because the original crimes data frame already contains a column named id. The function wants you to say how to discern them.

crimes %>% slice\_head(n = 3) %>%   
 select(id, street )

## # A tibble: 3 × 2  
## id street$id $name   
## <int> <int> <chr>   
## 1 116209089 1738379 On or near Parking Area  
## 2 116208620 1737636 On or near Supermarket   
## 3 116208619 1737385 On or near Kate Street

This will allow you duplicate column names:

crimes %>% slice\_head(n = 3) %>%   
 select(id, street, latitude) %>% unnest\_wider(col = street, names\_repair = "minimal")

## # A tibble: 3 × 4  
## id id name latitude   
## <int> <int> <chr> <chr>   
## 1 116209089 1738379 On or near Parking Area 52.637041  
## 2 116208620 1737636 On or near Supermarket 52.632052  
## 3 116208619 1737385 On or near Kate Street 52.634503

This creates somewhat odd column names, adding their index

crimes %>% slice\_head(n = 3) %>%   
 select(id, street, latitude) %>% unnest\_wider(col = street, names\_repair = "unique")

## New names:  
## • `id` -> `id...1`  
## • `id` -> `id...2`

## # A tibble: 3 × 4  
## id...1 id...2 name latitude   
## <int> <int> <chr> <chr>   
## 1 116209089 1738379 On or near Parking Area 52.637041  
## 2 116208620 1737636 On or near Supermarket 52.632052  
## 3 116208619 1737385 On or near Kate Street 52.634503

This will suffix the unnested column names with the data frame that they came from

crimes %>% slice\_head(n = 3) %>%   
 select(id, street, latitude ) %>%   
 unnest\_wider(col = street, names\_sep = "\_")

## # A tibble: 3 × 4  
## id street\_id street\_name latitude   
## <int> <int> <chr> <chr>   
## 1 116209089 1738379 On or near Parking Area 52.637041  
## 2 116208620 1737636 On or near Supermarket 52.632052  
## 3 116208619 1737385 On or near Kate Street 52.634503

You can paste a string before or after, write it like formulas in dplyr. For demonstration, I show only first three rows. Weirdly, unnest\_wider would throw an error with slice\_head(n = 3).

crimes %>% slice\_head(n = 3) %>%   
 select(id, street) %>%   
 unnest\_wider(col = street,   
 names\_repair = ~ paste(.x, c("\_crime", "\_street"), sep = ""))

## # A tibble: 3 × 3  
## id\_crime id\_street name\_crime   
## <int> <int> <chr>   
## 1 116209089 1738379 On or near Parking Area  
## 2 116208620 1737636 On or near Supermarket   
## 3 116208619 1737385 On or near Kate Street

Let’s apply names\_sep, which leaves the names of the original columns intact.

crimes <- crimes %>%   
 unnest\_wider(col = street,   
 names\_sep = "\_")

str(crimes$outcome\_status)

## 'data.frame': 1472 obs. of 2 variables:  
## $ category: chr NA NA NA NA ...  
## $ date : chr NA NA NA NA ...

It is again a data frame with one value per row of the crimes data frame, thus easy!

crimes %>% slice\_head(n = 10) %>% unnest\_wider(col = outcome\_status)

## Error in `unnest\_wider()`:  
## ! Can't duplicate names between the affected columns and the original  
## data.  
## ✖ These names are duplicated:  
## ℹ `category`, from `outcome\_status`.  
## ℹ Use `names\_sep` to disambiguate using the column name.  
## ℹ Or use `names\_repair` to specify a repair strategy.

The same story over! The first column of the original data frame has the same name.

crimes %>% slice\_head(n = 3)

## # A tibble: 3 × 12  
## category location\_type latitude street\_id street\_name longitude context  
## <chr> <chr> <chr> <int> <chr> <chr> <chr>   
## 1 anti-social-be… Force 52.6370… 1738379 On or near… -1.131222 ""   
## 2 anti-social-be… Force 52.6320… 1737636 On or near… -1.146096 ""   
## 3 anti-social-be… Force 52.6345… 1737385 On or near… -1.149403 ""   
## # ℹ 5 more variables: outcome\_status <df[,2]>, persistent\_id <chr>, id <int>,  
## # location\_subtype <chr>, month <chr>

Hence we have to deduplicate the column names again.

crimes %>% slice\_head(n = 10) %>%  
 unnest\_wider(col = outcome\_status,   
 names\_sep = "\_")

## # A tibble: 10 × 13  
## category location\_type latitude street\_id street\_name longitude context  
## <chr> <chr> <chr> <int> <chr> <chr> <chr>   
## 1 anti-social-b… Force 52.6370… 1738379 On or near… -1.131222 ""   
## 2 anti-social-b… Force 52.6320… 1737636 On or near… -1.146096 ""   
## 3 anti-social-b… Force 52.6345… 1737385 On or near… -1.149403 ""   
## 4 anti-social-b… Force 52.6350… 1737968 On or near… -1.139462 ""   
## 5 anti-social-b… Force 52.6274… 1737986 On or near… -1.138755 ""   
## 6 anti-social-b… Force 52.6275… 1738829 On or near… -1.124599 ""   
## 7 anti-social-b… Force 52.6267… 1738882 On or near… -1.123078 ""   
## 8 anti-social-b… Force 52.6378… 1738541 On or near… -1.130275 ""   
## 9 anti-social-b… Force 52.6240… 1739337 On or near… -1.113810 ""   
## 10 anti-social-b… Force 52.6237… 1737598 On or near… -1.146967 ""   
## # ℹ 6 more variables: outcome\_status\_category <chr>, outcome\_status\_date <chr>,  
## # persistent\_id <chr>, id <int>, location\_subtype <chr>, month <chr>

This works. So let’s save the change for the entire crimes data frame.

crimes <- crimes %>% unnest\_wider(col = outcome\_status,   
 names\_sep = "\_")

Check out DataCamp for the Reshaping Data with tidyr Course, Chapter **Rectangling Data** at <https://campus.datacamp.com/courses/reshaping-data-with-tidyr/rectangling-data?ex=1> to find out about more complex cases!