Lecture 14 - JavaScript Object Notation and APIs

Packages

```
# install packages
install.packages("jsonlite")
```

```
Installing package into '/usr/local/lib/R/site-library'
(as 'lib' is unspecified)
```

```
# load libraries
library(jsonlite)
```

JavaScript Object Notation (JSON)

- Most datasets are stored in tabular format using `.csv`, `.txt`, or `.xlsx` file types
- `R` loads these datasets as a data frame or "structured list"
- However, structured lists are highly inefficent at storing hierarchical or nested data structures

- For example, the data below shows a very simple nested structure stored in tabular form
- To accommdate the varying number of hobbies, we must repeat information across other columns
- Storing data in this way can be inefficient in terms of data storage

```
# load nested data in tabular format
```

```
read.csv("https://raw.githubusercontent.com/khasenst/datasets_teaching/refs/heads/main
         header = TRUE)
```

A data.frame: 5 × 6

name	age	address_street	address_city	address_zipcode	hobby
<chr>	<int>	<chr>	<chr>	<int>	<chr>
Alice	19	123 Main St	San Diego	92182	reading
Alice	19	123 Main St	San Diego	92182	hiking
Alice	19	123 Main St	San Diego	92182	surfing
Pedro	18	456 Elm St	Los Angeles	90745	gaming
Pedro	18	456 Elm St	Los Angeles	90745	cycling

- A more efficient way of storing nested data is JavaScript Object Notation (JSON)
- JSON format can be thought of as an unstructured (nested) list, where each list item can contain another data structure
  - i.e. a list of lists, data frames, vectors or a mixture of these

- The script below shows an example of JSON formatted data
- List items are defined within curly braces `{}`
- The colon `:` is followed by a data element
- The square brackets `[]` represent an array

```
# load nested file
nested <- fromJSON("https://raw.githubusercontent.com/khasenst/datasets_teaching/refs/

# print nested file
toJSON(nested, pretty = TRUE)
```

```
{
  "students": [
    {
      "name": "Alice",
      "age": 19,
      "address": {
        "street": "123 Main St",
        "city": "San Diego",
        "zipcode": "92182"
      },
      "hobbies": ["reading", "hiking", "surfing"]
    },
    {
      "name": "Pedro",
      "age": 18,
      "address": {
        "street": "456 Elm St",
        "city": "Los Angeles",
        "zipcode": "90745"
      },
      "hobbies": ["gaming", "cycling"]
    }
  ]
}
```

- In this format,
  - Activites are nested within "hobbies"
  - Address details are nested within "address"
  - All details are nested within "students"
- JSON format avoids the repetitive formatting of tabular representations of nested data

---

Loading JSON Data

- `R` loads JSON formatted data as unstructured lists [`list()`]
- We load JSON files using the `fromJSON()` function in the `jsonlite` library

```
# load json data
data <- fromJSON("https://raw.githubusercontent.com/khasenst/datasets_teaching/refs/he
```

- Remember, it is an unstructured list, so we can use the `$` operator to pull the nested data!

```
# json data is loaded as an unstructured list
class(data)
```

```
'list'
```

- To view the contents of the unstructured list, we can use the `str()` and `names()` functions

```
# view structure of json data in unstructured list format
str(data)
```

```
List of 1
 $ students:'data.frame':	2 obs. of  4 variables:
  ..$ name   : chr [1:2] "Alice" "Pedro"
  ..$ age    : int [1:2] 19 18
  ..$ address:'data.frame':	2 obs. of  3 variables:
  .. ..$ street : chr [1:2] "123 Main St" "456 Elm St"
  .. ..$ city   : chr [1:2] "San Diego" "Los Angeles"
  .. ..$ zipcode: chr [1:2] "92182" "90745"
  ..$ hobbies:List of 2
  .. ..$ : chr [1:3] "reading" "hiking" "surfing"
  .. ..$ : chr [1:2] "gaming" "cycling"
```

```
# using the names() function
names(data)
```

'students'

```
# using the names() function
names(data$students)
```

'name' · 'age' · 'address' · 'hobbies'

```
# using the $ to extract fields from the unstructured list
data$students$name
```

'Alice' · 'Pedro'

```
# pulling a field
data$students$hobbies
```

1. 'reading' · 'hiking' · 'surfing'
2. 'gaming' · 'cycling'

```
data$students$address
```

A data.frame: 2 × 3

	street	city	zipcode
	<chr>	<chr>	<chr>
1	123 Main St	San Diego	92182

1	123 Main St	San Diego	92182
2	456 Elm St	Los Angeles	90745

- Depending on how the JSON data is structured, the `fromJSON()` function attempts to simplify our data into a dataframe
- This only occurs if subfields do not further contains lists as indicated by curly braces `{}`

- For example, the new JSON formatted dataset now includes the address details as additional entries, as opposed to starting another list

```
# old format - address has a nested structure
'{
  "students": [
    {
      "name": "Alice",
      "age": 19,
      "address": {
        "street": "123 Main St",
        "city": "San Diego",
        "zipcode": "92182"
      },
      "hobbies": ["reading", "hiking", "surfing"]
    },
    {
      "name": "Pedro",
      "age": 18,
      "address": {
        "street": "456 Elm St",
        "city": "Los Angeles",
        "zipcode": "90745"
      },
      "hobbies": ["gaming", "cycling"]
    }
  ]
}'
```

```
'{\n  "students": [\n    {\n      "name": "Alice",\n      "age": 19,\n      "address": {\n        "street": "123 Main St",\n        "city": "San Diego",\n        "zipcode": "92182"\n      },\n      "hobbies": ["reading", "hiking", "surfing"]\n    },\n    {\n      "name": "Pedro",\n      "age": 18,\n      "address": {\n        "street": "456 Elm St",\n        "city": "Los Angeles",\n        "zipcode": "90745"\n      },\n      "hobbies": ["gaming",
```

```
# new json format without additional nesting for address
json_string <-
'{
 {

  "students": [
    {
      "name": "Alice",
      "age": 19,
      "address_street": "123 Main St",
      "address_city" : "San Diego",
      "address_zip" : "92182",
      "hobbies": ["reading", "hiking", "surfing"]
    },
    {
      "name": "Pedro",
      "age": 18,
      "address_street": "456 Elm St",
      "address_city": "Los Angeles",
      "address_zip": "90745",
      "hobbies": ["gaming", "cycling"]
    }
  ]

}'


# load the string from the JSON format
data2 <- fromJSON(json_string)


#  list output
str(data2)
```

```
List of 1
 $ students:'data.frame':	2 obs. of  6 variables:
  ..$ name          : chr [1:2] "Alice" "Pedro"
  ..$ age           : int [1:2] 19 18
  ..$ address_street: chr [1:2] "123 Main St" "456 Elm St"
  ..$ address_city  : chr [1:2] "San Diego" "Los Angeles"
  ..$ address_zip   : chr [1:2] "92182" "90745"
  ..$ hobbies       :List of 2
  .. ..$ : chr [1:3] "reading" "hiking" "surfing"
  .. ..$ : chr [1:2] "gaming" "cycling"
```

```
#  Now a data frame
data2$students
```

A data.frame: 2 × 6

	name	age	address_street	address_city	address_zip	hobbies
	<chr>	<int>	<chr>	<chr>	<chr>	<list>
1	Alice	19	123 Main St	San Diego	92182	reading, hiking , surfing
2	Pedro	18	456 Elm St	Los Angeles	90745	gaming , cycling

- We typically prefer the latter because the `fromJSON()` function does the work for us!
- If not, we must restructure our data using code to convert our data into a data frame

---

Application Program Interfaces (APIs)

- Given the efficiency of the JSON format for storing nested data, many companies make their data available as JSON files
- The way in which they make their data available is through an application program interface (API)

- An API is a set of rules that allows different software applications to communicate with each other for data transfer
- We can use an API to download data from an institution's servers to our R workspace!

- An example is the API for the World Health Organization (WHO)

  https://apps.who.int/gho/data/node.resources.api
- Different APIs have different rules for extracting data
- Institutions typically provide documentation on how to access their data
- We will focus on the WHO API as an example

Extracting Data using the WHO API

- Available "variables" are listed here: https://ghoapi.azureedge.net/api/
- One of the variables is life expectancy (at birth) `WHOSIS_000001`. Let's take a look!

- Loading the data on that variable

  https://ghoapi.azureedge.net/api/WHOSIS_000001

```
# path to data
url_path <- "https://ghoapi.azureedge.net/api/"

# selected variable
variable <- "WHOSIS_000001"

json_data <- fromJSON(paste0(url_path, variable),
                     #simplifyDataFrame = FALSE  # if you want it as a list, not a dat
                     )


# check contents of dataset
str(json_data)
```

```
List of 2
 $ @odata.context: chr "https://ghoapi.azureedge.net/api/$metadata#WHOSIS_000001"
 $ value         :'data.frame': 12936 obs. of  25 variables:
  ..$ Id                 : int [1:12936] 6548179 810 989 3390 6480 6743 7039 8253 8
  ..$ IndicatorCode      : chr [1:12936] "WHOSIS_000001" "WHOSIS_000001" "WHOSIS_00
  ..$ SpatialDimType     : chr [1:12936] "COUNTRY" "COUNTRY" "COUNTRY" "COUNTRY" ..
  ..$ SpatialDim         : chr [1:12936] "GTM" "SOM" "BTN" "BHR" ...
  ..$ TimeDimType        : chr [1:12936] "YEAR" "YEAR" "YEAR" "YEAR" ...
  ..$ ParentLocationCode: chr [1:12936] "AMR" "EMR" "SEAR" "EMR" ...
  ..$ ParentLocation     : chr [1:12936] "Americas" "Eastern Mediterranean" "South-
  ..$ Dim1Type           : chr [1:12936] "SEX" "SEX" "SEX" "SEX" ...
  ..$ TimeDim            : int [1:12936] 2020 2008 2002 2011 2005 2003 2014 2007 20
  ..$ Dim1               : chr [1:12936] "SEX_BTSX" "SEX_MLE" "SEX_BTSX" "SEX_FMLE"
  ..$ Dim2Type           : logi [1:12936] NA NA NA NA NA NA ...
  ..$ Dim2               : logi [1:12936] NA NA NA NA NA NA ...
  ..$ Dim3Type           : logi [1:12936] NA NA NA NA NA NA ...
  ..$ Dim3               : logi [1:12936] NA NA NA NA NA NA ...
  ..$ DataSourceDimType : logi [1:12936] NA NA NA NA NA NA ...
  ..$ DataSourceDim      : logi [1:12936] NA NA NA NA NA NA ...
  ..$ Value              : chr [1:12936] "71.0 [70.6-71.3]" "48.0 [46.7-49.6]" "67.
  ..$ NumericValue       : num [1:12936] 71 48 67.8 75.2 73.1 ...
  ..$ Low                : num [1:12936] 70.6 46.7 67.1 75.1 72.8 ...
  ..$ High               : num [1:12936] 71.3 49.6 68.6 75.4 73.5 ...
  ..$ Comments           : logi [1:12936] NA NA NA NA NA NA ...
  ..$ Date               : chr [1:12936] "2024-08-02T09:43:39.193+02:00" "2024-08-0
  ..$ TimeDimensionValue: chr [1:12936] "2020" "2008" "2002" "2011" ...
  ..$ TimeDimensionBegin: chr [1:12936] "2020-01-01T00:00:00+01:00" "2008-01-01T00
  ..$ TimeDimensionEnd  : chr [1:12936] "2020-12-31T00:00:00+01:00" "2008-12-31T00
```

```
# check headers of dataset
names(json_data)
```

```
'@odata.context' · 'value'
```

- For this particular API, the `value` list entry contains our data
- Thankfully, the `fromJSON` function was able to convert the JSON values into an `R` data frame...thank you WHO!

```
# view head of data frame
head(json_data$value, 3)

# store data
data <- json_data$value
```

```
        Id   IndicatorCode  SpatialDimType  SpatialDim  TimeDimType  ParentLocatic
     <int>           <chr>           <chr>       <chr>        <chr>
1  6548179  WHOSIS_000001         COUNTRY         GTM         YEAR
2      810  WHOSIS_000001         COUNTRY         SOM         YEAR
3      989  WHOSIS_000001         COUNTRY         BTN         YEAR
```

- Now that we have our dataset in a data frame in `R`, we can organize our data and do an analysis! We'll do this for our assignment.

Filtering/Subsetting via url

- The previous example shows how to import data in JSON format from an institutional

- The previous example shows how to import data in JSON format from an institutional website using their API

- But what if we don't want ALL of their data?
- What if their data is way too large for our purposes?

- Is there a way to select only what we need?
- Is there a way to download their data in pieces?
- Is there a way to subset on THEIR machines?

- The answer is Yes!
- Similar to filtering/subsetting in `R`, we can subset the data in the url itself

- For example, the script below downloads the same `WHOSIS_000001` dataset but with the following constraints
  - `SpatialDimType` must be `REGION`
  - `NumericValue` must not be missing (`NULL`)
  - `TimeDim` must be greater than or equal to `2020`

```
# root path to API
url_path <- "https://ghoapi.azureedge.net/api/"

# selected variable
variable <- "WHOSIS_000001"

# filter query
filter1 <- "?$filter=SpatialDimType%20eq%20'REGION'"
filter2 <- "%20and%20NumericValue%20ne%20null"
filter3 <- "%20and%20TimeDim%20ge%202020"


# query url
query_url <- paste0(url_path, variable, filter1, filter2, filter3)
query_url
```

```
'https://ghoapi.azureedge.net/api/WHOSIS_000001?
$filter=SpatialDimType%20eq%20\'REGION\'%20and%20NumericValue%20ne%20null%20and%20Ti
```

```
# import the data as an unstructured list
```

```
json_data <- fromJSON(query_url,
                     #simplifyDataFrame = FALSE  # if you want it as a list, not a dat
                     )
```

- Viewing the imported data, we see that our filtering/subsetting requirements are met

```
# the result
str(json_data)
```

```
List of 2
 $ @odata.context: chr "https://ghoapi.azureedge.net/api/$metadata#WHOSIS_000001"
 $ value         :'data.frame': 36 obs. of  25 variables:
  ..$ Id                : int [1:36] 744331 887617 1037630 2123034 2409991 2448337
  ..$ IndicatorCode     : chr [1:36] "WHOSIS_000001" "WHOSIS_000001" "WHOSIS_00000
  ..$ SpatialDimType    : chr [1:36] "REGION" "REGION" "REGION" "REGION" ...
  ..$ SpatialDim        : chr [1:36] "EUR" "AFR" "AFR" "AFR" ...
  ..$ TimeDimType       : chr [1:36] "YEAR" "YEAR" "YEAR" "YEAR" ...
  ..$ ParentLocationCode: logi [1:36] NA NA NA NA NA NA ...
  ..$ ParentLocation    : logi [1:36] NA NA NA NA NA NA ...
  ..$ Dim1Type          : chr [1:36] "SEX" "SEX" "SEX" "SEX" ...
  ..$ TimeDim           : int [1:36] 2021 2020 2021 2020 2020 2020 2021 2021 2021
  ..$ Dim1              : chr [1:36] "SEX_FMLE" "SEX_MLE" "SEX_BTSX" "SEX_FMLE" ..
  ..$ Dim2Type          : logi [1:36] NA NA NA NA NA NA ...
  ..$ Dim2              : logi [1:36] NA NA NA NA NA NA ...
  ..$ Dim3Type          : logi [1:36] NA NA NA NA NA NA ...
  ..$ Dim3              : logi [1:36] NA NA NA NA NA NA ...
  ..$ DataSourceDimType : logi [1:36] NA NA NA NA NA NA ...
  ..$ DataSourceDim     : logi [1:36] NA NA NA NA NA NA ...
  ..$ Value             : chr [1:36] "79.3 [79.2-79.5]" "62.0 [61.1-63.2]" "63.6 [
  ..$ NumericValue      : num [1:36] 79.3 62 63.6 65.9 68.9 ...
  ..$ Low               : num [1:36] 79.2 61.1 62.6 65.2 68.3 ...
  ..$ High              : num [1:36] 79.5 63.2 64.6 66.9 69.8 ...
  ..$ Comments          : logi [1:36] NA NA NA NA NA NA ...
  ..$ Date              : chr [1:36] "2024-08-02T09:43:39.193+02:00" "2024-08-02T0
  ..$ TimeDimensionValue: chr [1:36] "2021" "2020" "2021" "2020" ...
  ..$ TimeDimensionBegin: chr [1:36] "2021-01-01T00:00:00+01:00" "2020-01-01T00:00
  ..$ TimeDimensionEnd  : chr [1:36] "2021-12-31T00:00:00+01:00" "2020-12-31T00:00
```

- Instructions on how to do this are typically on the institution's website
- Note that in-depth API queries are outside the scope of this class

Exporting data as a JSON file

- We are able to export datasets as JSON files using
  - `toJSON()` - converts the data into a json class similar to character string
  - `write()` - then exports the character string using a given filename

- Let's do this with our initial example data

```
# load nested file
json_data <- fromJSON("https://raw.githubusercontent.com/khasenst/datasets_teaching/re

str(json_data)
```

```
List of 1
 $ students:'data.frame':	2 obs. of  4 variables:
  ..$ name   : chr [1:2] "Alice" "Pedro"
  ..$ age    : int [1:2] 19 18
  ..$ address:'data.frame':	2 obs. of  3 variables:
  .. ..$ street : chr [1:2] "123 Main St" "456 Elm St"
  .. ..$ city   : chr [1:2] "San Diego" "Los Angeles"
  .. ..$ zipcode: chr [1:2] "92182" "90745"
  ..$ hobbies:List of 2
  .. ..$ : chr [1:3] "reading" "hiking" "surfing"
  .. ..$ : chr [1:2] "gaming" "cycling"
```

- We then convert the unstructured list into a JSON formatted string

```
# convert to json string
json_data <- toJSON(json_data, pretty = TRUE)
json_data
```

```
{
  "students": [
    {
      "name": "Alice",
      "age": 19,
      "address": {
        "street": "123 Main St",
        "city": "San Diego",
        "zipcode": "92182"
      },
      "hobbies": ["reading", "hiking", "surfing"]
    },
    {
```

```
    {
      "name": "Pedro",
      "age": 18,
      "address": {
        "street": "456 Elm St",
        "city": "Los Angeles",
        "zipcode": "90745"
      },
      "hobbies": ["gaming", "cycling"]
    }
  ]
}
```

```
# class after toJSON()
class(json_data)
```

```
'json'
```

- Similar to `.csv` for comma separated value files, here, we use `.json` as the extension for JSON files
- The `write()` function is similar to the other exporting functions [`write.csv()`, `write.table()`], where you specify the data and the filepath

```
# export the json data structure to Colab
write(json_data, "student_data.json")
```