DSA2101

Essential Data Analytics Tools: Data Visualization

Yuting Huang

AY23/24 Semester 2

Weeks 3 – 4: Importing Data to `R`

Contents

In the next two weeks, we will learn how to import data into `R`.

1. CSV files
2. Flat files
3. Excel Files
4. R data files
5. JSON Files
6. Data from the Web

Recap

An important pre-requisite to loading data into `R` is that we are able to point to the location at which the data files are stored.

1. Where am I?
2. Where are my data?

Working directory

The first question addresses the notion of our current **working directory**.

- Typically, it is the location of our current `R` script.
- We can use the function `getwd()` to obtain the current working directory.

```
getwd()
```

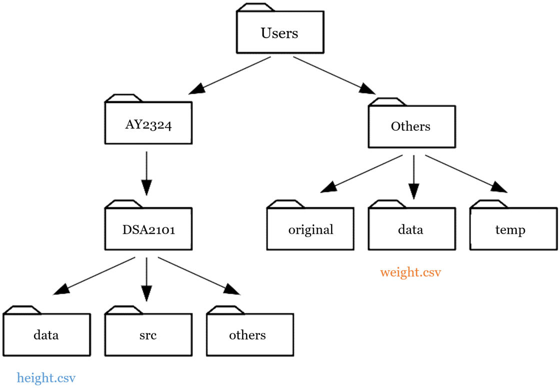
The function returns the **absolute path** of the current working directory.

File path

The second question implies that data are not necessarily stored at the location of our current working directory.

- Absolute path: the exact address of a file on our computer.
- **Relative path**: the address of a file relative to our current working directory.
  - Access files directly in the current working path.
  - Use two dots `..` to denote "one level up in the directory hierarchy".

Using relative path in all code you write. This allows you to share your scripts and data files easily with others.



Let's say your current working directory is
`C:/Users/AY2324/DSA2101/src`

- To access **height** data: `../data/height.csv`
- To access **weight** data: `../../../Others/data/weight.csv`

File path (Important!)

We will strictly follow the following practice:

- create a main folder titled **DSA2101**. Store all code, data, and markdown files inside.
- Within DSA2101, create a sub-folder called **src** to store all `R` scripts and `Rmd` files.
- Within DSA2101, create another sub-folder called **data** to store all data sets.
- The **src** and **data** folders are positioned at the same hierarchical level within **DSA2101**. **Use relative path in all code you write.**

Memory requirements for `R` objects

Before we read in data, remember that `R` stores all its objects using physical memory.

- It is important to be aware of how much memory is being used in your workspace.
- Especially when we are reading in or creating a new (large) data set in `R`.
  - It is often useful back-of-the-envelope calculation of how much memory the object will occupy in the `R` session.

Calculation

Suppose I have a data frame with 1,500,000 rows and 120 columns, all of which are numeric data.

- Roughly, on most modern computers, integers are 4 bytes, numerics are 8 bytes, and character data are usually 1 byte per character.
- Given that, we can do the following calculation.

$$
\begin{aligned}
1500000 \times 120 \times 8 \text{ bytes} &= 1440000000 \text{ bytes} \\
&= 1440000000/2^{20} \text{ MB} = 1373.29 \text{ MB} = 1.34 \text{ GB}
\end{aligned}
$$

- Most computers these days have at least that much of RAM, but you still need to be aware of
    - What other programs might be running on your computer, using up RAM, and
    - What other `R` objects might already be taking up RAM in your workspace.

Memory requirements for `R` objects

If you do not have enough RAM, your computer (or at least your `R` session) will freeze up.

- This is usually an unpleasant experience that requires you to kill the `R` session (the best scenario), or
- … reboot your computer (the worst case).

So make sure you understand the memory requirements before reading in or creating large data sets!

Read more about memory usage in RStudio on Posit.

CSV files

CSV stands for Comma-separated values.

- These files are in fact just text files, with
  - an optional header, listing the column names.
  - each observation separated by commas within each row
- CSV is the easiest format to read into R.

What does a CSV file look like?

A `.csv` file, opened in a text editor:

heights - Notepad

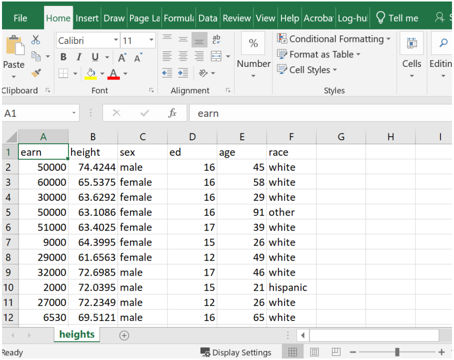
File Edit Format View Help

```
earn,height,sex,ed,age,race
50000,74.42443878,male,16,45,white
60000,65.53754283,female,16,58,white
30000,63.62919774,female,16,29,white
50000,63.10856168,female,16,91,other
51000,63.40248357,female,17,39,white
9000,64.39950754,female,15,26,white
29000,61.65632583,female,12,49,white
32000,72.69854374,male,17,46,white
2000,72.03946685,male,15,21,hispanic
27000,72.23493256,male,12,26,white
6530,69.51214643,male,16,65,white
30000,68.03160826,male,11,34,white
12000,67.55693392,male,12,27,white
12000,65.43058708,female,12,51,white
22000,65.66285452,female,16,35,white
17000,67.75876874,male,12,58,white
40000,68.35184258,female,14,29,white
44000,69.6095718,male,13,44,white
```

Ln 1, Col 1 | 100% | Windows (CRLF) | UTF-8

What does a CSV file look like?

Here is the same file opened in Excel:



earn	height	sex	ed	age	race
50000	74.4244	male	16	45	white
60000	65.5375	female	16	58	white
30000	63.6292	female	16	29	white
50000	63.1086	female	16	91	other
51000	63.4025	female	17	39	white
9000	64.3995	female	15	26	white
29000	61.6563	female	12	49	white
32000	72.6985	male	17	46	white
2000	72.0395	male	15	21	hispanic
27000	72.2349	male	12	26	white
6530	69.5121	male	16	65	white

Read a CSV file into `R`

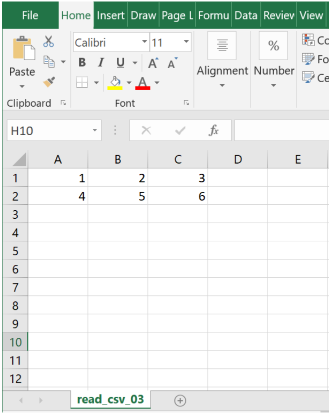
The command to read a CSV file into `R` is `read.csv()`

The main arguments to this function are:

- `file`: the file name.
- `header`: absence / presence of a header row.
- `skip`: number of lines at the beginning to skip.
- `col.names`: the names to identify columns in the table.
- `stringsAsFactors`: whether to convert character vectors to factors.
- `na.strings`: specify a character vector to be interpreted as `NA` values.

Example: A simple CSV file

- Take a first look at the data.
- 2 rows $\times$ 3 columns.
- The data set has no header.



Example: A simple CSV file

```r
df <- read.csv("../data/read_csv_03.csv", header = FALSE,
                col.names = c("a", "b", "c"))
df
```

```
##   a b c
## 1 1 2 3
## 2 4 5 6
```

- This file does not contain a header row, thus `header = FALSE`
- We can name the column as `a, b, c`. If we do not supply column names, R will name the columns by itself.

Example: Education, Height, and Income

`heights.csv` contains information on 1192 individuals.

- Take a look at the data, you will find that it contains 6 columns and 1 header.
- Hence, we read in the data in the following way:

```
heights <- read.csv("../data/heights.csv",
                    header = TRUE, stringsAsFactors = TRUE)
dim(heights)
```

```
## [1] 1192    6
```

- The function `dim()` (stands for **dimensions**) tells us that the data frame has 1192 rows and 6 columns.

Data checks

1. What type has each column been read in as?

```
str(heights)
```

```
## 'data.frame':    1192 obs. of  6 variables:
##  $ earn  : num  50000 60000 30000 50000 51000 9000 29000 32000 2000 27000 ..
##  $ height: num  74.4 65.5 63.6 63.1 63.4 ...
##  $ sex   : Factor w/ 2 levels "female","male": 2 1 1 1 1 1 1 2 2 2 ...
##  $ ed    : int  16 16 16 16 17 15 12 17 15 12 ...
##  $ age   : int  45 58 29 91 39 26 49 46 21 26 ...
##  $ race  : Factor w/ 4 levels "black","hispanic",..: 4 4 4 3 4 4 4 4 2 4 ...
```

- The function `str()` (stands for **structure**) reveals information about the columns, giving the names of the columns and a peek into the contents of each.
- We can see that the data types make sense.

Data checks

2. `race` is a categorical variable (a **factor** class in `R`). What are the different races that have been read in?

```
levels(heights$race)
```

```
## [1] "black"    "hispanic" "other"    "white"
```

- The function `levels()` returns the level of a factor variable.
- Recall that the dollar sign `$` extracts variable from a data frame.

Data checks

3. Are there any missing values in the data?

```
sum(is.na(heights))
```

```
## [1] 0
```

- Use `is.na()` to check missing entries in the entire data set.
- If there are missing values, we would also like to know which variable contains missing value.

```
apply(heights, 2, function(x) sum(is.na(x)))
```

```
##    earn height    sex     ed    age   race
##       0      0      0      0      0      0
```

Summary statistics

We can compute summary statistics for `earn`:

```
summary(heights$earn)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##     200   10000   20000   23155   30000  200000
```

Group statistics with `tapply()`:

```
tapply(heights$earn, heights$sex, mean)
```

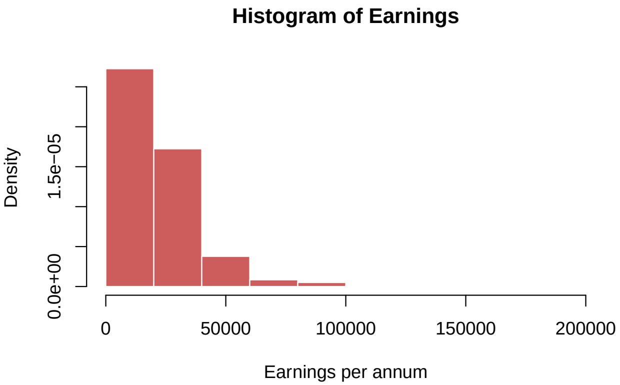
```
##   female     male
## 18280.20 29786.13
```

Histogram

Let us plot a histogram of income earned by individuals.

- A histogram divides the range of quantitative values into bins, then counts the number of values that fall into each bin.
- By default, the height of each bar represents frequencies.
- `freq = FALSE` alters a histogram such that the height represents the probability densities (that is, the histogram has a total area of one).

```r
hist(heights$earn, freq = FALSE,
     main = "Histogram of Earnings", xlab = "Earnings per annum",
     col = "indianred", border = "white")
```



- The distribution of income is right-skewed, as expected.

Histogram (revised code)

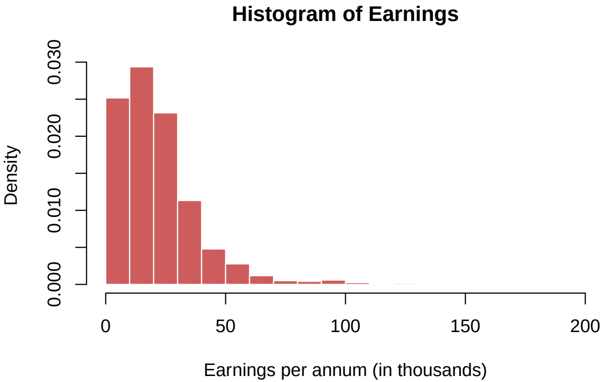
Our presentation of the histogram can be improved:

1. Disable scientific notations.
2. The bins correspond to intervals of width 20,000. Perhaps we would like bins of width 10,000 instead.

```
options(scipen = 999)
hist(heights$earn/1000, freq = FALSE,
     main = "Histogram of Earnings", xlab = "Earnings per annum (in thousands)"
     col = "indianred", border = "white", breaks = seq(0, 200, by = 10))
```

- `heights$earn/1000` divides earnings by a thousand. Now the earnings value ranges from 0 to 200.
- `breaks = seq(0, 200, by = 10)` sets the range of the x-axis from 0 to 200, and split it into bins with width 10.

Histogram (revised code)



The income distribution

Who are those high-earning individuals – earn more than 100K a year?

```r
# install.packages("tidyverse")
library(tidyverse)
filter(heights, earn > 1e5)
```

```
##     earn   height    sex ed age  race
## 1 125000 74.34062   male 18  45 white
## 2 170000 71.01003   male 18  45 white
## 3 175000 70.58955   male 16  48 white
## 4 148000 66.74020   male 18  38 white
## 5 110000 65.96504   male 18  37 white
## 6 105000 74.58005   male 12  49 white
## 7 123000 61.42908 female 14  58 white
## 8 200000 69.66276   male 18  34 white
## 9 110000 66.31203 female 18  48 other
```

The income distribution

The code on the previous slide uses the `tidyverse` syntax.

- It is an excellent tool for cleaning data.
- We shall study it very soon in Week 5.
- For now, only need to understand that it **filters out** irrelevant rows from the `heights` data frame, keeping only those who earned more than 10^5 per year.

Recap

- Remember that you should inspect your data before and after you read them in.
- Try to think of as many ways in which it could have gone wrong and check.
- As we covered here, you should at least consider the following:
  - Correct number of rows and columns.
  - Column variables read in with the correct class type.
  - Missing values.

Flat file

The `readr` package is developed to deal with reading in **large flat files** quickly.

- Much faster than base-R analogues, such as `read.csv()` or `read.table()`.
- We can use `read_csv()` to read in the `heights` data.
- A convenient argument is `col_types`, which specifies the type of each column.

```r
# install.packages("readr)
library(readr)
heights <- read_csv("../data/heights.csv", col_types = "infiif")
```

Other file types

`readr` provides other functions to read in data:

- `read_csv2()` reads semicolon-separated files.
- `read_tsv()` reads tab-delimited files.
- `read_delim()` reads in files with any delimiter, attempting to automatically guess the delimiter if you do not specify it.
- `read_fwf()` reads fixed-width files.
- ...

Useful documentation and cheatsheet on data import.

Excel files

To read data from `xls` and `xlsx` files, we need the `readxl` package.

```
# install.packages("readxl")
library(readxl)
```

- The `read_xlsx()` function automatically detects the rectangle region that contains non-empty cells in the Excel spreadsheet.
- Nonetheless, ensure that you open up your file in Excel first, to see what it contains and how you can provide further contextual information for the function to use.

Excel example

Let us see a simple example.

```
read_excel("../data/read_excel_01.xlsx")
```

```
## # A tibble: 7 x 5
##   `Table 1` ...2  ...3   ...4 ...5
##   <lgl>     <lgl> <chr> <dbl> <chr>
## 1 NA        NA    <NA>     NA <NA>
## 2 NA        NA    <NA>     NA <NA>
## 3 NA        NA    <NA>     NA <NA>
## 4 NA        NA    <NA>     NA <NA>
## 5 NA        NA    a         1 m
## 6 NA        NA    b         2 m
## 7 NA        NA    c         3 m
```

In this case, `read_excel()` needs a little help as the data seems to be "floating" in the center of the worksheet.

Excel example

```r
read_excel("../data/read_excel_01.xlsx", skip = 5)
```

```
## # A tibble: 2 x 3
##   a       `1` m
##   <chr> <dbl> <chr>
## 1 b         2 m
## 2 c         3 m
```

- The `skip` argument tells R to skip a certain number of rows.
- Looks like the function is reading the first row as the header. We can disable it by specifying `col_names = FALSE`.
- Notice that `read_excel()` uses a `col_names` argument, instead of `header`.

Excel example

Another way is the specify the data range exactly.

```
read_excel("../data/read_excel_01.xlsx", range = "C6:E8", col_names = FALSE)
```

```
## # A tibble: 3 x 3
##   ...1   ...2 ...3
##   <chr> <dbl> <chr>
## 1 a         1 m
## 2 b         2 m
## 3 c         3 m
```

- In case you were wondering, a `tibble` is an improved version of a data frame. We shall learn more about it in Week 5.

Example: UNESCAP data

The excel file `UNESCAP_population_2010_2015.xlsx` contains population counts for the Asia-Pacific countries.

- The counts are broken down by age group and gender.
- In the file, data for each age group and gender are stored in different spreadsheets.
- Suppose we want to read in data for **females aged 0–4 years**.

UNESCAP data on population

Read in data for **female aged 0–4 years old**.

```
female_0_4 <- read_excel("../data/UNESCAP_population_2010_2015.xlsx", sheet = 3
head(female_0_4)
```

```
## # A tibble: 6 x 7
##   e_fname     Y2010 Y2011 Y2012 Y2013 Y2014 Y2015
##   <chr>       <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
## 1 Afghanistan  2447  2459  2454  2438  2422  2412
## 2 Armenia        92    94    97    99   101   101
## 3 Australia     710   731   740   743   745   752
## 4 Azerbaijan    333   348   370   394   413   425
## 5 Bangladesh   7725  7622  7565  7540  7525  7503
## 6 Bhutan         35    35    35    34    33    32
```

To create a variable `age` based on the name of the spreadsheet.

```
library(stringr)   # for parsing sheet names

file_name <- "../data/UNESCAP_population_2010_2015.xlsx"
sheet_names <- excel_sheets(file_name)
female_0_4$age <- str_split(sheet_names[3], ",", simplify = TRUE)[3]
female_0_4$age <- str_trim(female_0_4$age)   # remove leading/ending space
head(female_0_4)
```

```
## # A tibble: 6 x 8
##   e_fname     Y2010 Y2011 Y2012 Y2013 Y2014 Y2015 age
##   <chr>       <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <chr>
## 1 Afghanistan  2447  2459  2454  2438  2422  2412 0-4 years
## 2 Armenia        92    94    97    99   101   101 0-4 years
## 3 Australia     710   731   740   743   745   752 0-4 years
## 4 Azerbaijan    333   348   370   394   413   425 0-4 years
## 5 Bangladesh   7725  7622  7565  7540  7525  7503 0-4 years
## 6 Bhutan         35    35    35    34    33    32 0-4 years
```

Read in data for **females aged 0–14 years**.

```
file_name <- "../data/UNESCAP_population_2010_2015.xlsx"
sheet_names <- excel_sheets(file_name)
sheet_names
```

```
## [1] "Pop- women"                "Pop- men"
## [3] "Pop, female, 0-4 years"    "Pop, female, 5-9 years"
## [5] "Pop, female, 10-14 years"  "Pop, Male, 0-4 years"
## [7] "Pop, Male, 5-9 years"      "Pop, Male, 10-14 years"
## [9] "Info"
```

```
sheet_names = sheet_names[3:5]
all_data = NULL    # create an empty object
for(names in sheet_names) {

  temp_data <- read_excel(file_name, sheet = names)
  age_grp <- str_split(names, ",", simplify = TRUE)[3]  # split a string
  temp_data$age <- str_trim(age_grp)       # remove leading/ending space
  all_data <- rbind(all_data, temp_data)   # bind by rows

}
```

Read in data for **females aged 0–14 years**.

```
all_data
```

```
## # A tibble: 150 x 8
##    e_fname           Y2010 Y2011 Y2012 Y2013 Y2014 Y2015 age
##    <chr>             <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <chr>
##  1 Afghanistan        2447  2459  2454  2438  2422  2412 0-4 years
##  2 Armenia              92    94    97    99   101   101 0-4 years
##  3 Australia           710   731   740   743   745   752 0-4 years
##  4 Azerbaijan          333   348   370   394   413   425 0-4 years
##  5 Bangladesh         7725  7622  7565  7540  7525  7503 0-4 years
##  6 Bhutan               35    35    35    34    33    32 0-4 years
##  7 Brunei Darussalam    15    14    15    16    16    16 0-4 years
##  8 Cambodia            817   839   851   858   862   868 0-4 years
##  9 China             36383 36577 37057 37721 38281 38538 0-4 years
## 10 DPR Korea           841   826   826   835   847   854 0-4 years
## # i 140 more rows
```

R data formats

R has two native data formats, Rds and Rdata.

- Rds is for a single R object. Rdata is used to save multiple R objects.
- Last time, we used a Rds file that contains a nested list.

```r
hawkers <- readRDS("../data/hawker_ctr_raw.rds")
str(hawkers[[1]][[7]])
```

```
## List of 12
##  $ ADDRESSBUILDINGNAME    : chr ""
##  $ ADDRESSFLOORNUMBER     : chr ""
##  $ ADDRESSPOSTALCODE      : chr "320091"
##  $ ADDRESSSTREETNAME      : chr "Whampoa Drive"
##  $ ADDRESSUNITNUMBER      : chr ""
##  $ DESCRIPTION            : chr "HUP Standard Upgrading"
##  $ HYPERLINK              : chr ""
##  $ NAME                   : chr "Blks 91/92 Whampoa Drive"
##  $ PHOTOURL               : chr ""
##  $ ADDRESSBLOCKHOUSENUMBER: chr "91/92"
##  $ XY                     : chr "30309.21,33962.7799"
##  $ ICON_NAME              : chr "HC icons_Opt 8.jpg"
```

Retrieving street names

Remove the first sub-list in `hawkers`

```
hawkers_116 <- hawkers[[1]][-1]
```

Name	Type	Value
hawkers	list [1]	List of length 1
SrchResults	list [117]	List of length 117
[[1]]	list [1]	List of length 1
[[2]]	list [12]	List of length 12
ADDRESSBUILDING...	character [1]	''
ADDRESSFLOORNU...	character [1]	''
ADDRESSPOSTALC...	character [1]	'141001'
ADDRESSSTREETN...	character [1]	'Commonwealth Drive'
ADDRESSUNITNU...	character [1]	''
DESCRIPTION	character [1]	'HUP Standard Upgrading'
HYPERLINK	character [1]	''
NAME	character [1]	'Blks 1A/ 2A/ 3A Commonwealth Drive'
PHOTOURL	character [1]	''
ADDRESSBLOCKHO...	character [1]	'1A/2A/3A'
XY	character [1]	'24055.5,31341.24'
ICON_NAME	character [1]	'HC icons_Opt 8.jpg'
[[3]]	list [12]	List of length 12

Name	Type	Value
hawkers_116	list [116]	List of length 116
[[1]]	list [12]	List of length 12
ADDRESSBUILDINGN...	character [1]	''
ADDRESSFLOORNUM...	character [1]	''
ADDRESSPOSTALCODE	character [1]	'141001'
ADDRESSSTREETNAME	character [1]	'Commonwealth Drive'
ADDRESSUNITNUMB...	character [1]	''
DESCRIPTION	character [1]	'HUP Standard Upgrading'
HYPERLINK	character [1]	''
NAME	character [1]	'Blks 1A/ 2A/ 3A Commonwealth Drive'
PHOTOURL	character [1]	''
ADDRESSBLOCKHOU...	character [1]	'1A/2A/3A'
XY	character [1]	'24055.5,31341.24'
ICON_NAME	character [1]	'HC icons_Opt 8.jpg'
[[2]]	list [12]	List of length 12
[[3]]	list [12]	List of length 12

Retrieving street names

The object **hawkers_116** contains 116 lists, each has 12 components.

- Retrieve the street names of the first component with the following

```
hawkers_116[[1]]$ADDRESSSTREETNAME
```

```
## [1] "Commonwealth Drive"
```

- The following code produces the same output.

```
hawkers_116[[1]][[4]]
```

```
## [1] "Commonwealth Drive"
```

Retrieving street names

To retrieve all street names, use `sapply()` with an anonymous function to store them in a vector.

```
street_name <- sapply(hawkers_116, function(x) x$ADDRESSSTREETNAME)
head(street_name, n = 10)
```

```
##  [1] "Commonwealth Drive"     "Marsiling Lane"          "Boon Lay Place"
##  [4] "Havelock Road"          "Circuit Road"            "Whampoa Drive"
##  [7] "Upper Bukit Timah Road" "Smith Street"            "Kensington Park Road
## [10] "Yishun Ring Road"
```

Converting to a data frame

- Using the same trick on different components in the sub-list, we can store variables in different vectors.
- Then we can combine them as a new data frame.

```
postal_code <- sapply(hawkers_116, function(x) x$ADDRESSPOSTALCODE)
name <- sapply(hawkers_116, function(x) x$NAME)
coordinates <- sapply(hawkers_116, function(x) x$XY)

hawkers_df <- data.frame(postal_code, name, coordinates)
head(hawkers_df, n = 4)
```

```
##   postal_code                                 name         coordinates
## 1      141001 Blks 1A/ 2A/ 3A Commonwealth Drive    24055.5,31341.24
## 2      730020           Blks 20/21 Marsiling Lane   21755.23,47282.71
## 3      641221          Blks 221A/B Boon Lay Place 14587.57,36373.7899
## 4      161022            Blks 22A/B Havelock Road  27589.1399,30043.3
```

Converting to a data frame

It is still inconvenient to extract data from the sub-lists one by one.

- We can convert the entire list into a data frame. In the following code,
  - `lapply()` to convert sub-lists to individual lists.
  - Each list contains a data frame with 1 row and 12 columns.
  - `do.call()` and `rbind()` to combine the list of data frames by rows.

```
hawkers_df <- do.call(rbind, lapply(hawkers_116, as.data.frame))
```

```r
hawkers_df <- do.call(rbind, lapply(hawkers_116, as.data.frame))
str(hawkers_df)
```

```
## 'data.frame':    116 obs. of  12 variables:
##  $ ADDRESSBUILDINGNAME    : chr  "" "" "" "" ...
##  $ ADDRESSFLOORNUMBER     : chr  "" "" "" "" ...
##  $ ADDRESSPOSTALCODE      : chr  "141001" "730020" "641221" "161022" ...
##  $ ADDRESSSTREETNAME      : chr  "Commonwealth Drive" "Marsiling Lane" "Boon
##  $ ADDRESSUNITNUMBER      : chr  "" "" "" "" ...
##  $ DESCRIPTION            : chr  "HUP Standard Upgrading" "HUP Standard Upgr
##  $ HYPERLINK              : chr  "" "" "" "" ...
##  $ NAME                   : chr  "Blks 1A/ 2A/ 3A Commonwealth Drive" "Blks
##  $ PHOTOURL               : chr  "" "" "" "" ...
##  $ ADDRESSBLOCKHOUSENUMBER: chr  "1A/2A/3A" "20/21" "221A/B" "22A/B" ...
##  $ XY                     : chr  "24055.5,31341.24" "21755.23,47282.71" "145
##  $ ICON_NAME              : chr  "HC icons_Opt 8.jpg" "HC icons_Opt 8.jpg" "
```

- The nested list is now converted to a data frame with 116 observations and 12 variables.

JavaScript Object Notation (JSON)

JSON (JavaScript Object Notation) is a standard **text-based format** for storing structured data.

- On the internet, it is a very popular format for data interchange.
- The full description of the format can be found at http://www.json.org/
- The syntax is easy for humans to read and write, and for computers to parse and generate.

We shall work with the `jsonlite` package.

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

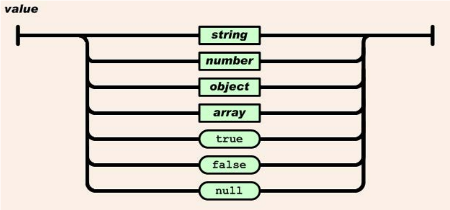
JSON description

- JSON is built on two structures:
  - An **object** is an unordered collection of name/value pairs.
  - An **array** is an ordered list of values.
- By repeatedly stacking these structures on top of one another, we will be able to store quite complex data structures.

```
object
    {}
    { members }
members
    pair
    pair , members
pair
    string : value
array
    []
    [ elements ]
elements
    value
    value , elements
value
    string
    number
    object
    array
    true
    false
    null
```

JSON value

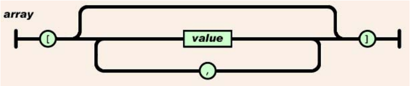
A **value** can be a string (in double quotes), a number, an object, an array, or a true or false or null.



JSON array

An **array** is an ordered collection of values.

- Surrounded with square brackets, starts with `[` and ends with `]`
- Values are separated by a comma `,`



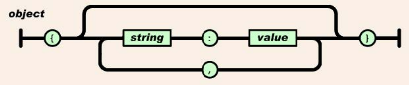
Example:

- `[12, 3, 7]` is an JSON array with three elements, all are numbers.
- `["Hello", 3, 7]` is also valid.

JSON object

An **object** is an unordered set of name/value pairs.

- Surrounded with curly braces, starts with `{` and ends with `}`
- Each name is followed by a colon `:` and the name/value pairs are separated by a comma `,`



Example:

- `{"fruit": "Apple"}` is a valid JSON object.
- `{"fruit": "Apple", "price": 2.03}` is also valid.
  - Two name/value pairs. The names are "fruit" and "price".

Read JSON objects in R

The `fromJSON()` function in the `jsonlite` package allows us to read JSON from files, the web, or straight from the console.

1. In the following example, `fromJSON()` detects that the values are homogeneous and so reads them into a numeric vector.

```
txt <- "[12, 3, 7]"
fromJSON(txt)
```

```
## [1] 12  3  7
```

Read JSON objects in R

2. In this case, the values are not all of the same type. So the function reads them in as a character vector.

```
txt2 <- '[12, "a", 7]'
fromJSON(txt2)
```

```
## [1] "12" "a"  "7"
```

3. The missing value is coded as `NA`.

```
txt3 <- '[12, null, 7]'
fromJSON(txt3)
```

```
## [1] 12 NA  7
```

Read a single JSON object from a file

- JSON stores everything as a text.
- If we are sure that the `txt` file only contains one JSON object, we can use the command `fromJSON()`.

```
fromJSON("../data/read_json_01.txt")
```

```
## $fruit
## [1] "Apple"
##
## $price
## [1] 2.03
##
## $shelf
## [1] "lower"  "middle"
```

Read multiple JSON objects from a file

- If the file has multiple JSON objects, we need to first read each line into R using `readLines()`, and then apply `fromJSON()` to each of them.

```
all_lines <- readLines("../data/read_json_02.txt")
json_list <- lapply(all_lines, fromJSON)
str(json_list)
```

```
## List of 3
##  $ :List of 3
##   ..$ fruit: chr "Apple"
##   ..$ price: num 2.03
##   ..$ shelf: chr [1:2] "lower" "middle"
##  $ :List of 3
##   ..$ fruit: chr "Orange"
##   ..$ price: num 1.03
##   ..$ shelf: chr [1:2] "middle" "upper"
##  $ :List of 3
##   ..$ fruit: chr "Watermelon"
##   ..$ price: num 0.99
##   ..$ shelf: chr "lower"
```

Convert to a data frame

The next step is to convert it into a data frame.

- Notice that watermelons can only be stored on the lower shelf, but the other two fruits can be stored in two possible shelves.
- How should the data frame look like?

fruit	**price**	**shelf**
Apple	2.03	lower, middle
Orange	1.03	middle, upper
Watermelon	0.99	lower

OR

fruit	**price**	**lower**	**middle**	**upper**
Apple	2.03	1	1	0
Orange	1.03	0	1	1
Watermelon	0.99	1	0	0

Convert to a data frame

Let us first write a function (`convert_2_df`) that takes one component at a time and then converts it to a data frame.

```
convert_2_df <- function(x) {

  lower  = ifelse("lower"  %in% x$shelf, 1, 0)
  middle = ifelse("middle" %in% x$shelf, 1, 0)
  upper  = ifelse("upper"  %in% x$shelf, 1, 0)

  data.frame(fruit = x$fruit, price = x$price, lower, middle, upper)
}
```

Convert to a data frame

Apply this new function `convert_2_df` to the list `json_list` to obtain a **list** of three data frames.

```
df_row <- lapply(json_list, convert_2_df)
df_row
```

```
## [[1]]
##   fruit price lower middle upper
## 1 Apple  2.03     1      1     0
##
## [[2]]
##    fruit price lower middle upper
## 1 Orange  1.03     0      1     1
##
## [[3]]
##        fruit price lower middle upper
## 1 Watermelon  0.99     1      0     0
```

Convert to a data frame

We then combine these individual rows into one single data frame using `rbind()`.

```
df_fruit <- rbind(df_row[[1]], df_row[[2]], df_row[[3]])
df_fruit
```

```
##        fruit price lower middle upper
## 1      Apple  2.03     1      1     0
## 2     Orange  1.03     0      1     1
## 3 Watermelon  0.99     1      0     0
```

Example: New York Restaurant Scores

New York consists of 5 boroughs - Bronx, Brooklyn, Manhattan, Queens, and Staten Island.

- The data set `restaurants_dataset.json` contains inspection results of restaurants in New York.
  - 25359 restaurants in total.
  - Most restaurants are inspected more than once. Each inspection gives a letter grade and a numeric violation score.

Read in data

Given the data, we would like to

- Compute the average violation score for each restaurant.
- compute the mean of the average violation score for each borough.

Let's first read in the JSON file.

```
NYrest <- readLines("../data/restaurants_dataset.json")
NYrest_json <- lapply(NYrest, fromJSON)
length(NYrest_json)
```

```
## [1] 25359
```

- It is a list of 25359 elements.

```r
str(NYrest_json[[1]])
```

```
## List of 6
##  $ address      :List of 4
##   ..$ building: chr "1007"
##   ..$ coord   : num [1:2] -73.9 40.8
##   ..$ street  : chr "Morris Park Ave"
##   ..$ zipcode : chr "10462"
##  $ borough      : chr "Bronx"
##  $ cuisine      : chr "Bakery"
##  $ grades       :'data.frame':   5 obs. of  3 variables:
##   ..$ date : POSIXct[1:5], format: "2014-03-03 08:00:00" "2013-09-11 08:00:0
##   ..$ grade: chr [1:5] "A" "A" "A" "A" ...
##   ..$ score: int [1:5] 2 6 10 9 14
##  $ name         : chr "Morris Park Bake Shop"
##  $ restaurant_id: chr "30075445"
```

- The `grades` component contains a column named `score`. This is what we are after.
- Observe that this particular restaurant has been inspected 5 times.

Inspect the data set

Some thoughts before we proceed to analyze the data:

1. Are all boroughs represented in the data?
2. Are the `restaurant_id`s unique?
3. Do all restaurants have at least one score? Are there restaurants with missing scores?

1. Are all boroughs represented in the data?
    - Count the number of restaurants in each borough.

```
all_borough <- sapply(NYrest_json, function(x) x$borough)
table(all_borough)
```

```
## all_borough
##         Bronx      Brooklyn     Manhattan       Missing        Queens
##          2338          6086         10259            51          5656
## Staten Island
##           969
```

- Remove the 51 observations with missing borough information.

```
id <- which(all_borough == "Missing")
NYrest_json <- NYrest_json[-id]
length(NYrest_json)
```

```
## [1] 25308
```

2. Are the `restaurant_ids` unique?

```
all_rest_ids <- sapply(NYrest_json, function(x) x$restaurant_id)
length(unique(all_rest_ids))
```

```
## [1] 25308
```

- The `unique()` function extracts unique elements from a vector or a data frame.
- The restaurant id's are unique.

3. Do all restaurants have at least one score?
    - Check if all restaurants have been inspected at least once.

```
n_scores <- sapply(NYrest_json, function(x) nrow(x$grades))
head(n_scores, n = 3)
```

```
## [[1]]
## [1] 5
##
## [[2]]
## [1] 4
##
## [[3]]
## [1] 4
```

- Why does it return a list? `sapply()` should return a vector.
- I conclude that not all elements in `grades` are data frames. Let's check.

```r
for (i in 1:length(NYrest_json)) {
  c1 <- class(NYrest_json[[i]]$grades)
    if (c1 != "data.frame") {
      print(paste("The first anomaly at position", i))
      break
  }
}
```

```
## [1] "The first anomaly at position 15310"
```

```r
class(NYrest_json[[i]]$grades)
```

```
## [1] "list"
```

```r
length(NYrest_json[[i]]$grades)
```

```
## [1] 0
```

- This restaurant has an **empty list** instead of a data frame.
- We need to weed these cases out!

- Need to have a little more intelligence to count the number of inspection scores (instead of just counting the number of rows in the `grades` data frame).

```
score_count <- function(x) {
  if (class(x$grades) == "data.frame") {
    scores <- nrow(x$grades)
  } else {
    scores <- 0
  }
return(scores)
}
n_scores <- sapply(NYrest_json, score_count)
sum(n_scores == 0)
```

```
## [1] 737
```

Thus, from the inspections, there are 737 restaurants without a single rating.

- Let's remove these observations before proceeding.

```
NYrest_json <- NYrest_json[n_scores != 0]
length(NYrest_json)
```

```
## [1] 24571
```

- There are 24571 remaining restaurants to work with.

Mean violation scores

What is the average violation score for each restaurant?

```
mean_score <- sapply(NYrest_json, function(x) mean(x$grades$score))
sum(is.na(mean_score))
```

```
## [1] 13
```

- After computing the mean, as a pre-cautious step, I checked whether there are any missing mean values.
- Turns out there are still 13 `NA` returned!
- It means that some restaurants had no score, but were **not** indicated by a list.
- We need to take a closer look…

```r
# Locate the NA mean scores
id <- which(is.na(mean_score))
# Run the following command in your Console.
# This checks the individual components in the "grade" element.
# lapply(NYrest_json[id], function(x) x$grades)
```

- We will remove these entries.
- In fact, upon further inspection, there is also one restaurant with a negative score. We will remove that as well.

```r
id1 <- which(mean_score < 0)
lapply(NYrest_json[id1], function(x) x$grades)
```

```
## [[1]]
##                  date grade score
## 1 2014-11-13 08:00:00     B    -1
```

Remove these problematic observations.

```
id2 <- append(id, id1)
NYrest_json <- NYrest_json[-id2]
length(NYrest_json)
```

```
## [1] 24557
```

- After cleaning up, we are left with 24557 observations.

Analysis

- Finally, compute mean score for each restaurant and summarize:

```
mean_score <- sapply(NYrest_json, function(x) mean(x$grades$score))
summary(mean_score)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##   0.000   8.333  10.429  11.123  13.000  75.000
```

- Mean scores for each borough.

```
borough <- sapply(NYrest_json, function(x) x$borough)
tapply(mean_score, borough, mean)
```

```
##         Bronx      Brooklyn     Manhattan        Queens Staten Island
##      10.87371      11.09892      11.06674      11.34453      11.18433
```

Summary

We must inspect a data set to detect anomalies before proceeding to data analysis.

Here is a break down of what we did at each step.

- We start from the original data set with 25359 observations.
- Remove 51 observations with missing `borough`. 25308
- Remove 737 observations whose `score` is a list. 24571
- Remove 13 observations whose `score` is an empty data frame, and 1 observation with negative `socre`. 24557

Finally, the analysis is done on a clean data set with 24557 observations.

Data from the web

We can read data files directly from a website to `R`.

TidyTuesday is a weekly social data project in `R` born out of the *R for Data Science* textbook and its online learning community.

- It posts raw data set(s) and a related article every week.
- Emphasizes on the understanding of how to summarize and arrange data to make meaningful visuals in the `tidyverse` ecosystem.

Full list of data sets can be found on

https://github.com/rfordatascience/tidytuesday

TidyTuesday data

Let's explore the data set posted on April 20, 2021.

- A data set on TV shows and movies available on Netflix.
- You can find an overview of the data at:

https://github.com/rfordatascience/tidytuesday/blob/master/data/2021/2021-04-20/readme.md

Follow the instruction to get the data.

- Method 1: Read in the data with the `tidytuesdayR` package (daily limit applies).

```
# install.packages("tidytuesdayR")
tuesdata <- tidytuesdayR::tt_load("2021-04-20")
netflix <- tuesdata$netflix_titles
```

- Method 2: Read in data manually via URL.

```
netflix <- readr::read_csv('https://raw.githubusercontent.com/rfordatascience/t
head(netflix, 4)
```

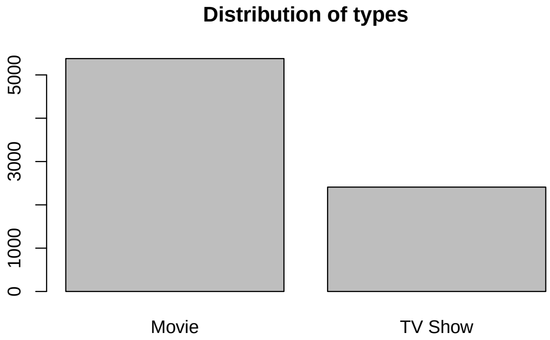
```
## # A tibble: 4 x 12
##   show_id type    title director     cast  country date_added release_year ra
##   <chr>   <chr>   <chr> <chr>        <chr> <chr>   <chr>             <dbl> <c
## 1 s1      TV Show 3%    <NA>         João~ Brazil  August 14~         2020 TV
## 2 s2      Movie   7:19  Jorge Mich~ Demi~ Mexico  December ~         2016 TV
## 3 s3      Movie   23:59 Gilbert Ch~ Tedd~ Singap~ December ~         2011 R
## 4 s4      Movie   9     Shane Acker Elij~ United~ November ~         2009 PG
## # i 3 more variables: duration <chr>, listed_in <chr>, description <chr>
```

```r
summary(netflix)
```

```
##    show_id              type              title             director        
##  Length:7787        Length:7787        Length:7787        Length:7787       
##  Class :character   Class :character   Class :character   Class :character  
##  Mode  :character   Mode  :character   Mode  :character   Mode  :character  
##                                                                             
##                                                                             
##                                                                             
##      cast             country           date_added         release_year 
##  Length:7787        Length:7787        Length:7787        Min.   :1925  
##  Class :character   Class :character   Class :character   1st Qu.:2013  
##  Mode  :character   Mode  :character   Mode  :character   Median :2017  
##                                                           Mean   :2014  
##                                                           3rd Qu.:2018  
##                                                           Max.   :2021  
##     rating            duration          listed_in         description       
##  Length:7787        Length:7787        Length:7787        Length:7787       
##  Class :character   Class :character   Class :character   Class :character  
##  Mode  :character   Mode  :character   Mode  :character   Mode  :character  
##                                                                             
##                                                                             
## 
```

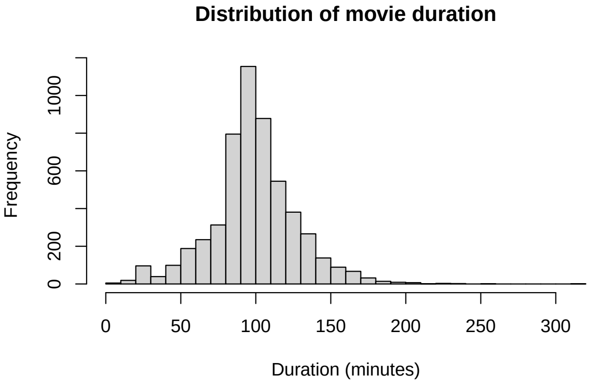
A bar plot on the types of Netflix titles.

```r
netflix$type <- as.factor(netflix$type)
barplot(table(netflix$type), main = "Distribution of types")
```

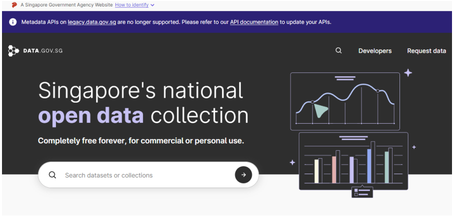


A histogram of movie duration.

```r
movie <- netflix[netflix$type == "Movie", ]
movie$minute <- as.numeric(stringr::str_replace(movie$duration, " min", ""))
hist(movie$minute, breaks = 30,
     xlab = "Duration (minutes)", main = "Distribution of movie duration")
```



data.gov.sg

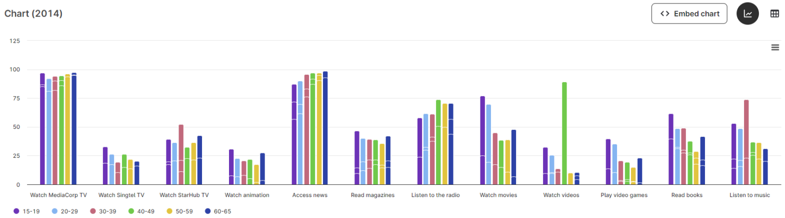


- data.gov.sg was launched in 2011 as Singapore's national open data portal.
- Data sets from 70 government agencies, in the fields of economy, education, environment, finance, health, infrastructure, etc.
- From the website, data sets can be downloaded in `csv` format.
- It is also possible to download the data using a script. The data would then be returned as a `JSON` object.

Media usage data from IMDA

Every year, the Infocomm Media Development Authority (IMDA) commissions a Media Consumer Experience Study.

- The data describe the percentage of consumers who have ever used a traditional media device (e.g., TV, newspaper) for media activities.



Download IMDA data

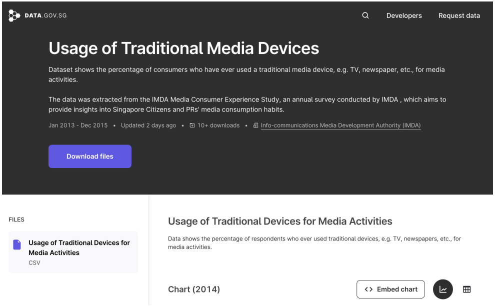
Now we demonstrate how to download the data through `R`.

- Instructions on querying the data through API can be found in the **Developers** sub-page on the website.
- Essentially what is needed is to identify the **resource id** for this data set, and then tag it onto a template URL.
  - The URL from the data set page shows the resource id for this data.
- However, there is a limit on the number of records that can be retrieved per query. Thus it is necessary to run a loop until all records have been retrieved.

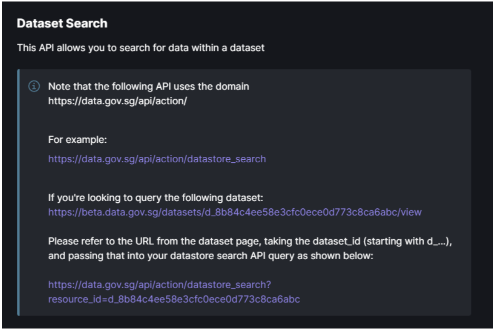
- Visit the web page:

  https://beta.data.gov.sg/collections/226/view

- Then click **Developers** on the upper right corner.



- This will direct you to the **API v2** documentation.
- Scroll down to the **Dataset Search** section.



- Examine the URL to the IMDA data. Find that the **resource id** for it is `d_1a88e269bf1d629b93fb5cafa189f9fb`.

- Now we will pass it to our API query.

```
url <- paste0("https://data.gov.sg",
              "/api/action/datastore_search?",
              "resource_id=d_1a88e269bf1d629b93fb5cafa189f9fb")
results_json <- fromJSON(url)
```

- The `fromJSON()` command coverts JSON format into a list.
- Now we can see that the `results_json` object is a list of length 3.

```r
str(results_json)
```

```
## List of 3
##  $ help   : chr "https://data.gov.sg/api/3/action/help_show?name=datastore_s
##  $ success: logi TRUE
##  $ result :List of 5
##   ..$ resource_id: chr "d_1a88e269bf1d629b93fb5cafa189f9fb"
##   ..$ fields     :'data.frame':  6 obs. of  2 variables:
##   .. ..$ type: chr [1:6] "numeric" "text" "text" "numeric" ...
##   .. ..$ id  : chr [1:6] "year" "age" "media_activity" "sample_size" ...
##   ..$ records    :'data.frame':  100 obs. of  6 variables:
##   .. ..$ _id           : int [1:100] 1 2 3 4 5 6 7 8 9 10 ...
##   .. ..$ year          : chr [1:100] "2013" "2013" "2013" "2013" ...
##   .. ..$ age           : chr [1:100] "15-19" "15-19" "15-19" "15-19" ...
##   .. ..$ media_activity: chr [1:100] "Watch MediaCorp TV" "Watch Singtel TV"
##   .. ..$ sample_size   : chr [1:100] "161" "161" "161" "161" ...
##   .. ..$ ever_used     : chr [1:100] "97.1" "32.9" "39.5" "30.9" ...
##   ..$ _links     :List of 2
##   .. ..$ start: chr "/api/action/datastore_search?resource_id=d_1a88e269bf1d
##   .. ..$ next : chr "/api/action/datastore_search?resource_id=d_1a88e269bf1d
##   ..$ total      : int 210
```

Download IMDA data

The list structure tells us that data are stored in the `results` sub-list.

- `results -> records`: We have managed to retrieve a data frame with 100 rows.
- `results -> total`: The final data set should contain 210 rows.

```
results_json[["result"]][["total"]]
```

```
## [1] 210
```

- `results -> _links`: The link we need to submit another query.

```
results_json[["result"]][["_links"]]
```

```
## $start
## [1] "/api/action/datastore_search?resource_id=d_1a88e269bf1d629b93fb5cafa189
##
## $`next`
## [1] "/api/action/datastore_search?resource_id=d_1a88e269bf1d629b93fb5cafa189
```

- Read the second component of `_links`.
- It tells us to offset the first 100 rows in the **next** query.

Download IMDA data

- Continue to submit queries **until** the requisite number of rows (= 210) are obtained.

```r
# Current number of rows
results_data <- results_json[["result"]][["records"]]

# Total expected number of rows
total_records <- results_json[["result"]][["total"]]

# Queries
while(nrow(results_data) < total_records) {
  url1 <- paste0("https://data.gov.sg",
                 results_json[["result"]][["_links"]][["next"]])
  results_json <- fromJSON(url1)
  results_data <- rbind(results_data, results_json[["result"]][["records"]])
}
```

To confirm that we have the data now:

```
str(results_data)
```

```
## 'data.frame':    210 obs. of  6 variables:
##  $ _id           : int  1 2 3 4 5 6 7 8 9 10 ...
##  $ year          : chr  "2013" "2013" "2013" "2013" ...
##  $ age           : chr  "15-19" "15-19" "15-19" "15-19" ...
##  $ media_activity: chr  "Watch MediaCorp TV" "Watch Singtel TV" "Watch StarH
##  $ sample_size   : chr  "161" "161" "161" "161" ...
##  $ ever_used     : chr  "97.1" "32.9" "39.5" "30.9" ...
```

Plotting IMDA data

Let us make a bar chart for the 20-29 years age group.

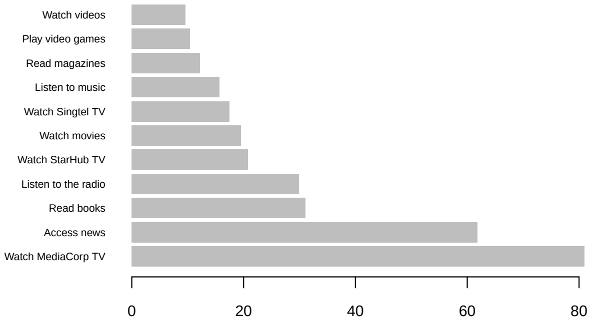
- The following code will become comprehensible after the next lecture. For now, you only need to understand its purpose:
  - Filter and keep the rows we want.
  - Convert the variable `ever_used` from character to numeric, save it as `pct`.
  - Sort the data set by descending order of `pct`.

```
library(tidyverse)
young <- filter(results_data, age == "20-29", year == 2015) %>%
  mutate(pct = as.numeric(ever_used)) %>%
  arrange(desc(pct))
head(young, n = 2)
```

```
##    _id year   age     media_activity sample_size ever_used  pct
## 1 156 2015 20-29 Watch MediaCorp TV         395        81 81.0
## 2 161 2015 20-29        Access news         395      61.8 61.8
```

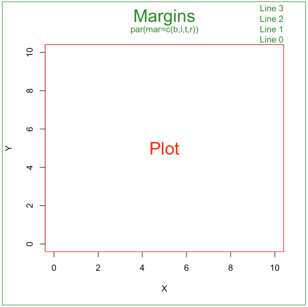
- Alter the arguments below and study their effects on the plot.

```r
par(mar = c(5, 7, 2, 2))
barplot(young$pct,
        names.arg = young$media_activity,
        horiz = TRUE, las = 1, cex.names = 0.6, cex.axis = 0.8, border = NA)
```



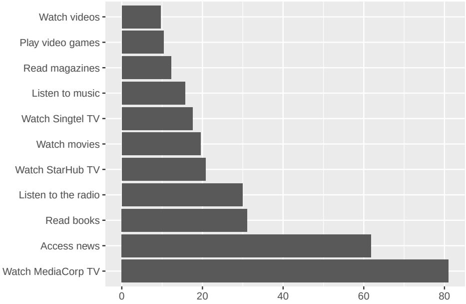
Adjusting the margins of the plot

- `par(mar = c(5, 7, 4, 2))` on the previous slide specifies the margins on the four sides of the plot
- The default is `c(5, 4, 4, 2)`



The `ggplot()` way

```
ggplot(data = young, aes(x = reorder(media_activity, -pct), y = pct)) +
  geom_bar(stat = "identity") +
  coord_flip() + labs(x = "", y = "")
```



- Later in the semester, we shall learn about graphing with `ggplot()` functions.

Your turn

Data on Graduate Employment Survey.

https://beta.data.gov.sg/collections/415/view

- Tweak our code to download the data from data.gov.sg

```
str(results_data)
```

```
## 'data.frame':    1121 obs. of  13 variables:
##  $ _id                     : int  1 2 3 4 5 6 7 8 9 10 ...
##  $ year                    : chr  "2013" "2013" "2013" "2013" ...
##  $ university              : chr  "Nanyang Technological University" "Nanya
##  $ school                  : chr  "College of Business (Nanyang Business Sc
##  $ degree                  : chr  "Accountancy and Business" "Accountancy (
##  $ employment_rate_overall : chr  "97.4" "97.1" "90.9" "87.5" ...
##  $ employment_rate_ft_perm : chr  "96.1" "95.7" "85.7" "87.5" ...
##  $ basic_monthly_mean      : chr  "3701" "2850" "3053" "3557" ...
##  $ basic_monthly_median    : chr  "3200" "2700" "3000" "3400" ...
##  $ gross_monthly_mean      : chr  "3727" "2938" "3214" "3615" ...
##  $ gross_monthly_median    : chr  "3350" "2700" "3000" "3400" ...
##  $ gross_mthly_25_percentile: chr  "2900" "2700" "2700" "3000" ...
##  $ gross_mthly_75_percentile: chr  "4000" "2900" "3500" "4100" ...
```

Summary

We learn about importing data from different formats and sources:

1. CSV file using `read.csv()`
2. Flat file using functions from the `readr` package.
3. Excel file with `read_excel()` from the `readxl` package.
4. R data file with `readRDS()`.
5. JSON file with `fromJSON()` from the `jsonlite` package.
6. Data from the web.

Also a few more ways to clean and visualize data.

Summary

- Importing data becomes complicated when data is not stored in a friendly format.
- When reading data from the web, we need to have some creativity to identify patterns or keywords that can be used in a loop.
- The paths and patterns are unlikely to be the same every time, but the experience you gather will help you along.