Importing Data Into R

Vik Gopal

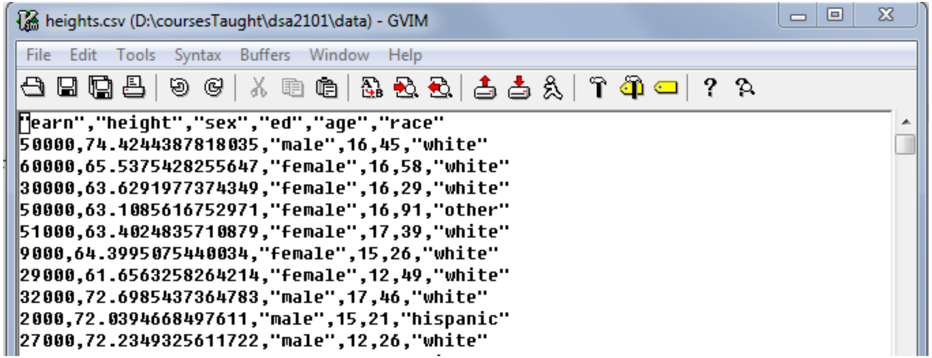
Let the programmers be many and the managers few,
– then all will be productive.

Comma-Separated Values

- CSV stands for **Comma-Separated Values**.
- These files are in fact just text files, with
  - An optional header, listing the column names.
  - Observations separated by commas within each row.
- It is important to remember this because, sometimes, it is easier to clean the data in our text editor than R.
- CSV is in fact the easiest format to read into R.
- Excel workbooks can be exported to csv format easily, and then imported into R. Soon, we shall introduce a package that works with Excel files directly.

What Does a CSV File Look Like?

Here is a csv file, opened in a text editor:

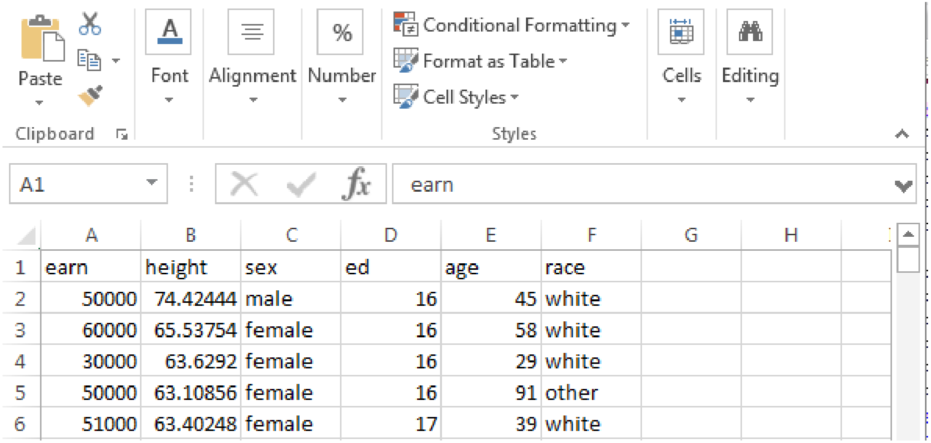

heights.csv (D:\coursesTaught\dsa2101\data) - GVIM
File Edit Tools Syntax Buffers Window Help

```
"earn","height","sex","ed","age","race"
50000,74.4244387818035,"male",16,45,"white"
60000,65.5375428255647,"female",16,58,"white"
30000,63.6291977374349,"female",16,29,"white"
50000,63.1085616752971,"female",16,91,"other"
51000,63.4024835710879,"female",17,39,"white"
9000,64.3995075440034,"female",15,26,"white"
29000,61.6563258264214,"female",12,49,"white"
32000,72.6985437364783,"male",17,46,"white"
2000,72.0394668497611,"male",15,21,"hispanic"
27000,72.2349325611722,"male",12,26,"white"
```

What Does a CSV File Look Like?

cont'd

Here is the same file, in Excel:



	A	B	C	D	E	F	G	H
1	earn	height	sex	ed	age	race		
2	50000	74.42444	male	16	45	white		
3	60000	65.53754	female	16	58	white		
4	30000	63.6292	female	16	29	white		
5	50000	63.10856	female	16	91	other		
6	51000	63.40248	female	17	39	white		

Reading 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 filename.
  - `header:` Absence/presence of a header row.
  - `skip:` Number of comment lines at the beginning.
  - `row.names:` The names to be used to identify rows in the table. This could even be one of the columns in the table.
  - `stringsAsFactors:` Whether to read strings in as factors or not.
- If successful, it will return a data frame containing the data.

A Rough Guide

- It is always a good idea to view your csv file in a text editor before reading it into R.
  - If it's not possible to load the whole file into your text editor, at least peruse the first few lines.
  - This will give you some indication of the amount of metadata, the presence/absence of headers, and how many columns there are in your data.
  - If you can, make a mental note of how many lines there are in the csv file - this corresponds to the number of observations there should be in the resulting data frame.
- After reading it into R, inspect at least the first and last few observations of the data frame. Check the following:
  - Were the correct number of rows read in?
  - Were the column names and column classes correctly assigned?
  - Were the missing values correctly read in?

Reading in Some Simple Files

- For `read.csv()`, the default value for the `header` argument is `TRUE`.
- Hence we did not need to specify it here.

```r
data_1 <- read.csv("../data/read_csv_01.csv")
data_1
```

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

Reading in Some Simple Files

cont'd

- Since the first two lines in the file are comments, we have to tell R to skip them.
- As a result of the command above, R will look for the header values in row 3.

```
data_2 <- read.csv("../data/read_csv_02.csv", skip=2)
data_2
```

```
  x y z
1 1 2 3
```

Reading in Some Simple Files

cont'd

- This file did not contain a header row.
- Hence, we have to tell R that this is so, and then specify the column names that we want.
  - If we do not supply `col.names`, R will name the columns by itself.

```
data_3 <- read.csv("../data/read_csv_03.csv",
                   header=FALSE,
                   col.names=c("a", "b", "c"))
data_3
```

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

Education, Height and Income

- This dataset contains data on 1192 individuals.
- If you take a look at the file, you will find that it contains 6 columns, a header, and no meta-data.
- Hence we read in the data in the following way:

```r
heights <- read.csv("../data/heights.csv")
```

- We then proceed with some basic data checks.

Data Checks

- What type has each column been read 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   : chr  "male" "female" "female" "female" ...
$ 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  : chr  "white" "white" "white" "other" ...
```

- We can see that the data types make sense.

Data Checks

cont'd

- What are the different races that have been read in? Sometimes, a missing value might be read in as a new race.

```
unique(heights$race)
```

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

- Are there missing values in any column? We can use `apply()` with an anonymous function for this.
  - Remember to apply it column-wise!

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

Income Distribution

code

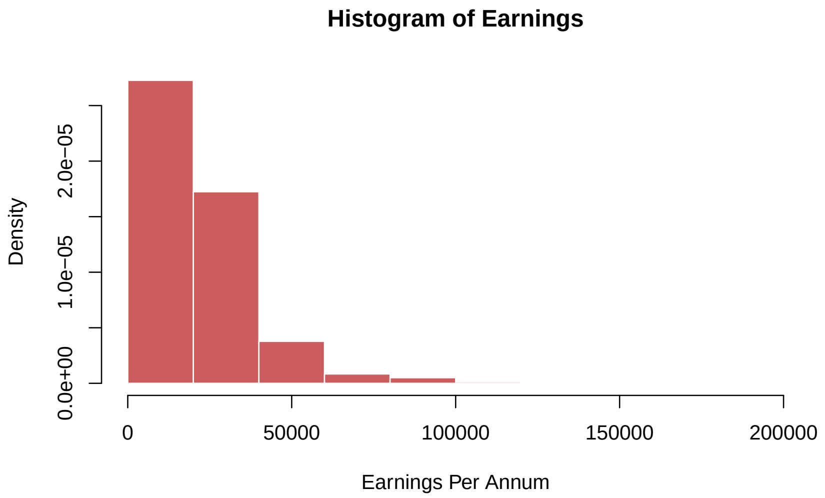
- Let us plot a histogram of the income earned by the individuals in the survey.

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

- A histogram divides the range of values into bins, then counts the number of values that fall into each bin.
- The `freq=FALSE` option alters the histogram such that the height of each bar does not represent a count.
- Instead, it is the height such that the area of all bars adds up to 1, making a histogram closer in spirit to a pdf (probability density function).

Income Distribution

plot



Income Distribution

cont'd

- The distribution of incomes is right-skewed, as expected.
- Although we do not know more about the origin of the dataset, our presentation of the histogram can improve:
  - The breaks, or bins, correspond to intervals of width 20,000. Perhaps we would like bins of width 10,000 instead.
  - Who are those high-earning individuals - those earning more than $100,000 a year?

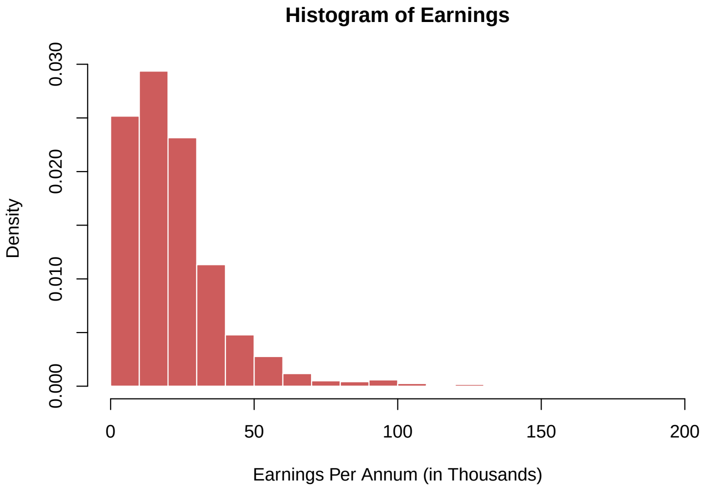
Income Distribution

revised code

```
hist(heights$earn/1000, main="Histogram of Earnings",
     freq=FALSE,
     xlab="Earnings Per Annum (in Thousands)",
     col="indianred", border="white",
     breaks=seq(0, 200, by=10))
```

Income Distribution

revised plot



Income Distribution

cont'd

- Let's take a look at those high-earners in a little more detail.

```
heights[heights$earn > 1e5, ]
```

```
       earn   height    sex ed age  race
175  125000 74.34062   male 18  45 white
203  170000 71.01003   male 18  45 white
340  175000 70.58955   male 16  48 white
357  148000 66.74020   male 18  38 white
377  110000 65.96504   male 18  37 white
567  105000 74.58005   male 12  49 white
751  123000 61.42908 female 14  58 white
1078 200000 69.66276   male 18  34 white
1184 110000 66.31204 female 18  48 other
```

Income Distribution

summary

- From the new histogram, it is easier to tell, for instance, that more than 50% of the individuals earned less than 20K per year. This calculation comes from

$$0.025 \times 10 + 0.03 \times 10$$

- More than 90% of them earned less than 50K per year.
- From inspecting the outliers,
  - Only two females earned more than 100K per year, compared to 7 males.
  - None of those earning more than 100K were black or hispanic.
  - The highest earner is also the youngest guy in that group!

English Premier League (EPL) Data

- The file `EPL_1415_1516.csv` contains information on matches played in the English Premier League for 2 seasons: 14/15 and 15/16.
- There were 20 teams in the league in each of those seasons.
- The csv file contains the following details about each of the 760 matches played:
  - Date of match,
  - Home team
  - Away team,
  - Referee
  - Timings of each goal scored in a game.
  - Which team scored each goal in a game.

EPL Data

column description

- Take a look at the csv file and try to match the headers to the variables listed on the previous slide.
- For instance,
  - `FGT` refers to First Goal Timing
  - `SGT` refers to Second Goal Timing, etc.
  - `FGW` stands for Who scored the First Goal. A value of `H` means that the home team scored it, and a value of `A` means that the away team scored it.
- In the event that a particular goal did not happen in a game, the corresponding columns will contain a hyphen `"-"`.
- For instance if a match ended `2-1`, then the columns `FOGT`, `FIGT`,... and `FOGW`, `FIGW`,.. will all contain a hyphen.

EPL Data

reading in data

- Here is our first attempt to read in the data.

```
epl <- read.csv("../data/EPL_1415_1516.csv",
                header=TRUE)
str(epl)
```

```
'data.frame': 760 obs. of  22 variables:
 $ Date    : chr  "16/8/2014" "16/8/2014" "16/8/2014" "16/8/2014" ...
 $ HomeTeam: chr  "Arsenal" "Leicester" "Man United" "QPR" ...
 $ AwayTeam: chr  "Crystal Palace" "Everton" "Swansea" "Hull" ...
 $ Referee : chr  "J Moss" "M Jones" "M Dean" "C Pawson" ...
 $ FGT     : chr  "35" "20" "28" "52" ...
 $ SGT     : chr  "45" "22" "53" "-" ...
 $ TGT     : chr  "90" "45" "72" "-" ...
 ...
```

EPL Data

problems

- All columns have been read in as character, because none of them are purely numeric.
- Let us use all the information we have to read in the data more precisely.

```
epl <- read.csv("../data/EPL_1415_1516.csv",
                na.strings = "-", header=TRUE)
# Try str(epl) now
```

- It is still not ideal, since we want the goal scorers columns to be factors.

EPL Data

cont'd

- This code specifies the class of each column.

```
epl <- read.csv("../data/EPL_1415_1516.csv",
                na.strings = "-", header=TRUE,
                colClasses = c(rep("character", 4),
                               rep("integer", 9),
                               rep("factor", 9)))
```

- Just a couple of sticky points:
  - The first column should be a `Date`.
  - The `NGW` column only has 1 level - this is because there was no game where the home team scored the 9th goal.
- We can rectify these manually now.

EPL Data

manual fix

- Here is how we can fix the Date column:

```
epl$Date <- as.Date(epl$Date, format='%d/%m/%Y')
str(epl$Date)
```

```
 Date[1:760], format: "2014-08-16" "2014-08-16" "2014-08-16" ...
```

- Here is how we can fix the *NGW* column issue:

```
epl$NGW <- factor(epl$NGW, levels=c('A', 'H'))
str(epl$NGW)
```

```
 Factor w/ 2 levels "A","H": NA NA NA NA NA NA NA NA NA NA ...
```

- We shall return to this dataset in the next topic and re-arrange it into a tidier format.

When `read.csv()` Fails

- When no amount of finicking with `read.csv()` arguments gets the data in correctly, you can still get the data into R as a character vector.
- We can then parse the strings using `str_detect_all()`, `str_split()`, and so on.
- The command used in this case is `readLines()`.

```r
f1_lines <- readLines("../data/read_csv_02.csv")
f1_lines
```

```
[1] "The first line of metadata"
[2] "The second line of metadata"
[3] "x,y,z"
[4] "1,2,3"
```

- Remember that this will work easiest when your data is text data, not binary.

Re-cap

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

Package readxl

- The readxl package provides a very powerful function for reading data from xls and xlsx files.
- As usual, the author (Hadley Wickham) has built a great deal of intelligence and intuition into the function.
  - It automatically detects the region that contains data in the spreadsheet.
  - It makes an educated guess as to the type of data stored in the cell.
- Nonetheless, ensure that you open up your file first, to see what it contains and how you can provide further contextual information for the function to use.
- Ensure that you have installed and loaded the library before proceeding with the lecture code:

```
library(readxl)
```

Data Rectangle Detection

The extent of the data rectangle can be detected in the following ways:

- It can be **discovered**. `read_excel()` uses the smallest rectangle that contains the non-empty cells.
- The number of rows that it searches through can be **bounded** through the use of the following two arguments:
  - `skip` tells it to skip a certain number of rows.
  - `n_max` tells it to look only up to a certain maximum number of rows.
- The precise region to use can be **set** using the `range` argument. In this case, empty cells will be filled with NA.

Excel Example

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

```
# A tibble: 8 x 5
  `Table 1` X__1  X__2   X__3 X__4
  <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    NA       NA NA
6 NA        NA    a         1 m
7 NA        NA    b         2 m
8 NA        NA    c         3 m
```

- Here is an example where the library is not able to discover the data by itself.
- The data seems to be “floating” in the centre of the worksheet.

Excel Example

cont'd

```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
```

- It looks like the function is reading in the first row as the header.
- Watch out! `read_excel()` uses a `col_names` argument, not `header`.
- In case you were wondering, a `tibble` is an improved version of a data frame. We shall learn more about it in topic 03.

Excel Example

cont'd

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

```
read_excel("../data/read_excel_01.xlsx",
              range="C7:E9",
              col_names=FALSE)
```

- This works correctly.
- An alternative, shown below, is to specify the data range exactly.

More Information on `read_excel()`.

- For more information on the `read_excel()` function, take a look at the vignettes that are shipped with the package:

```
vignette('sheet-geometry')
vignette('cell-and-column-types')
```

- Before we proceed with one last example with Excel, it is worth highlighting one of the quotes from the vignette:

  Cells you can see don't necessarily exist. Cells that look blank aren't necessarily so.

UNESCAP Data on Population

- UNESCAP is a commission under the United Nations that studies social and economic conditions in the Asia-Pacific region.
- 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.
- Suppose we wish to read in the male and female groups for 0 - 14 years and combine them into one dataset.

UNESCAP Data on Population

females 0 to 4 years old

```
fname <- "../data/UNESCAP_population_2010_2015.xlsx"
female_0_4 <- read_excel(fname, 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
```

UNESCAP Data on Population

cont'd

- We have read in only a single spreadsheet.
- However, there are multiple spreadsheets in the document.
- How can we combine them into one dataset?

UNESCAP Data on Population

cont'd

- These are the names of all the excel sheets in the spreadsheet.

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

UNESCAP Data on Population

cont'd

```r
library(stringr)          # For parsing sheet names

sheet_names <- sheet_names[3:8]

all_data <- NULL          # An empty object

for(sn in sheet_names){
  tmp_data <- read_excel(fname, sheet=sn)
  sn2 <- str_trim(str_split(sn, ",")[[1]])
  tmp_data$gender <- str_to_title(sn2[2])
  tmp_data$age_group <- sn2[3]
  all_data <- rbind(all_data, tmp_data)
}
```

UNESCAP Data on Population

cont'd

```
all_data
```

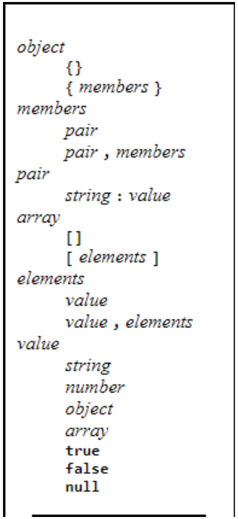
```
# A tibble: 300 x 9
             e_fname Y2010 Y2011 Y2012 Y2013 Y2014 Y2015 gender age_group
               <chr> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>  <chr>     <chr>
 1       Afghanistan  2447  2459  2454  2438  2422  2412 Female 0-4 years
 2           Armenia    92    94    97    99   101   101 Female 0-4 years
 3         Australia   710   731   740   743   745   752 Female 0-4 years
 4        Azerbaijan   333   348   370   394   413   425 Female 0-4 years
 5        Bangladesh  7725  7622  7565  7540  7525  7503 Female 0-4 years
 6            Bhutan    35    35    35    34    33    32 Female 0-4 years
 7 Brunei Darussalam    15    14    15    16    16    16 Female 0-4 years
 8          Cambodia   817   839   851   858   862   868 Female 0-4 years
 9             China 36383 36577 37057 37721 38281 38538 Female 0-4 years
10         DPR Korea   841   826   826   835   847   854 Female 0-4 years
# ... with 290 more rows
```

JavaScript Object Notation (JSON)

- JSON is a **text 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.
  - There are several R libraries that can generate and parse JSON files - `rjson`, `RJSONIO` and `jsonlite`.
  - We shall work with the `jsonlite` package.

JSON Description

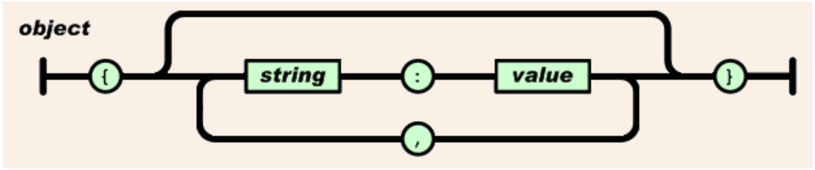
- JSON is built on two structures:
  - An unordered collection of name/value pairs. This is an **object**.
  - An ordered list of values. This is referred to as an **array**.
- By repeatedly stacking these structures on top of one another, we will be able to store quite complex data structures.



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

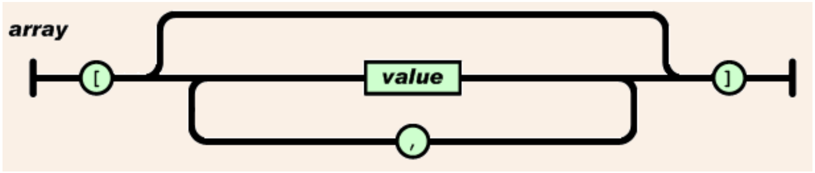
JSON Object

- A JSON object is an unordered set of name/value pairs.
- An object begins with { and ends with }.
- Each name is followed by : and the name/value pairs are separated by ,



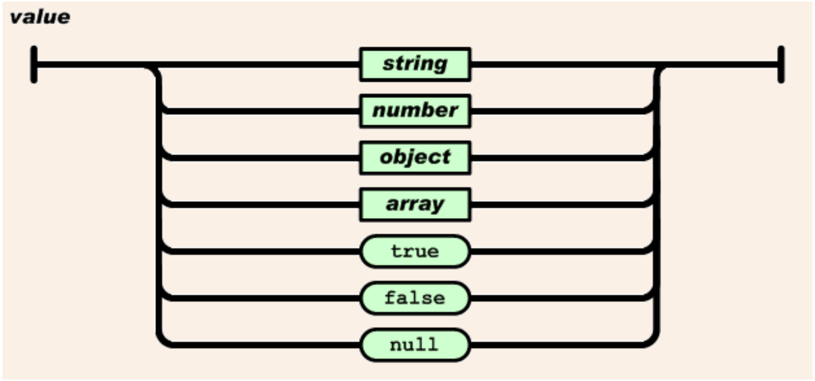
JSON Array

- A JSON array is an ordered collection of values.
- An array begins with `[` and ends with `]`.
- Values are separated by `,`



JSON value

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



JSON Examples

- `[12, 3, 7]` is a valid JSON structure.
  - It consists of a single array.
  - The elements are comprised of 3 values, all of which are numbers.
- `{"fruit": "Apple", "price": 2.03, "shelf": ["lower", "middle"]}` is a valid JSON object.
  - There are 3 name/value pairs. The names are "fruit", "price" and "shelf".
  - The value in the 3rd pair is an array, with 2 elements, which are both strings.
- `[12, 'a', 7]` is not a valid JSON array. Why?
- `[12, "a", 7]` is a valid JSON array.
- `[12, null, 7]` is also valid. The null value usually represents values that are missing or unset.

Reading JSON Objects Into R

- The `jsonlite` package allows one to read JSON objects from files, the web, or even straight from the console.

```r
library(jsonlite)
txt <- '[12, 3, 7]'
j1 <- fromJSON(txt)
j1
```

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

- The package detects that the values are homogeneous and so reads them into a numeric vector.

Reading JSON Objects Into R

cont'd

- In this case, since the values are not all of the same type, `jsonlite` reads them in as a character vector.

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

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

- Missing values in JSON strings will be appropriately coded as NA within R.

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

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

Reading A Single JSON Object From a File

- Recall that JSON stores everything as a text (not binary), and hence we can inspect it with our favourite text editor.
- If we are sure that there is only one JSON object, we can use the same command as earlier.

```
fruit <- fromJSON("../data/read_json_01.txt")
```

Pretty Printing of JSON String

- In R, if we have a JSON string, we can print it with indentation for easier understanding.

```r
fruit_string <- readLines("../data/read_json_01.txt")
prettify(fruit_string)
```

```
{
    "fruit": "Apple",
    "price": 2.03,
    "shelf": [
        "lower",
        "middle"
    ]
}
```

Reading Multiple JSON Objects From a File

- In this case, we first read each line into R, then apply `fromJSON()` to each of them.

```r
all_lines <- readLines("../data/read_json_02.txt")
json_list <- lapply(all_lines, fromJSON)
str(json_list, max.level = 2)
```

```
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"
 ...
```

Converting to A Data Frame

- How can we convert this information on fruits 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.
- Let us first write a function that takes one component and converts it to a data frame

```
comp_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)
}
```

Converting to A Data Frame

cont'd

- Applying this new function to the recently constructed list will yield a list of data frame rows.
- We then call upon `do.call()` to combine these individual rows into one single data frame.

```
df_row_list <- lapply(json_list, comp_2_df)
tidy_fruit <- do.call("rbind", df_row_list)
# rbind(df_row_list[[1]], df_row_list[[2]],
#       df_row_list[[3]])

tidy_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
```

New York Restaurant Scores

- New York consists of 5 boroughs - Bronx, Brooklyn, Manhattan, Queens, and Staten Island.
- The dataset that we are going to work with contains ratings of restaurants in New York.
  - Most restaurants are rated more than once.
- How do the ratings in the different boroughs relate to one another?
- For each borough, we would like to:
  - Compute the average score for each restaurant there.
  - Compute the mean of the average scores for each borough.

New York Restaurant Scores

Inspecting the Dataset

- From below, we see that the grades component is a data frame, which contains a column named scores - that is what we are after.
- Observe that this restaurant has been reviewed 5 times.

```
fname <- "../data/restaurants_dataset.json"
rest_lines <- readLines(fname)
rest_json <- lapply(rest_lines, fromJSON)
str(rest_json[[1]], max.level = 1)
```

```
List of 6
 $ address      :List of 4
 $ borough      : chr "Bronx"
 $ cuisine      : chr "Bakery"
 $ grades       :'data.frame':   5 obs. of  3 variables:
 $ name         : chr "Morris Park Bake Shop"
 $ restaurant_id: chr "30075445"
```

New York Restaurant Scores

Inspecting the Dataset

- Some thoughts before proceeding:
  - Are all boroughs represented?
  - Are the restuarant_id's unique?
  - Do all restaurants have at least one score? Are there restaurants with missing scores?
- Take note that there are 25359 records in the dataset.

New York Restaurant Scores

boroughs

- Here is a count of the number of restaurants in each borough.

```
all_bor <- sapply(rest_json, function(x) x$borough)
table(all_bor)
```

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

New York Restaurant Scores

restaurant ids

- The check below shows that restaurant ids are unique.
- There are 51 restaurants with missing borough information. We will not be able to use them here.

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

```
[1] 25359
```

New York Restaurant Scores

at least one inspection

- Now let us check if all restaurants have been inspected at least once.

```
n_scores <- sapply(rest_json,
                   function(x) NROW(x$grades))
sum(n_scores == 0)
```

```
738
```

- Using `NROW( )` instead of `nrow( )` ensures that a 0 (instead of `NULL`) is returned if there was no data frame for a particular restaurant.

New York Restaurant Scores

missing observations

- Thus we have 51 restaurants without a borough, and 738 without a single rating.
- We would like to remove these observations before proceeding.

```r
borough_missing <- str_detect(all_bor, "Missing")
score_missing <- n_scores == 0
rest_json_no_missing <- rest_json[!(borough_missing |
                                        score_missing)]
length(rest_json_no_missing)
```

```
[1] 24571
```

- There are 24571 remaining restaurants to work with.

New York Restaurant Scores

try for mean scores

- We proceed to compute the mean scores by borough.

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

```
[1] 13
```

- What?! There are still 13 NA's returned.
- It looks like some restaurants had no score, but this was not indicated by an empty list.
- Let's take a closer look.

New York Restaurant Scores

clean-up

- The following command selects those with NA mean scores, and checks what they contain in the `grades` component.

```
id <- which(is.na(mean_score))
# lapply(rest_json_no_missing[id],
#         function(x) x$grades)
```

- Thankfully, the boroughs have no further problems.

```
bor <- sapply(rest_json_no_missing,
              function(x) x$borough)
sum(is.na(bor))
```

```
[1] 0
```

New York Restaurant Scores

mean scores by borough

- Finally, here are the mean scores.

```
tapply(mean_score[-id],bor[-id], mean)
```

```
    Bronx      Brooklyn     Manhattan      Queens  Staten Island
 10.87371      11.09892      11.06553    11.34453       11.18433
```

- `tapply` is a cousin of `sapply`, that applies a function on a vector after dividing it into groups. We will have little use for it once we learn about the `dplyr` verbs in topic 03.

Object Oriented Programming (OOP) Models in R

- There are three different implementations of OOP within R.
- The simplest and most commonly used one is known as S3. It is easy to use, perhaps too easy.
- The next is S4. It is more rigorous than the S3 method.
- The implementations of S3 and S4 are a little different from the OOP you would have encountered in Python. The methods do not belong to the class; instead, we define *generic* functions, and then write specific methods for each class.
- R dispatches the appropriate method every time the generic function is called.
- The most sophisticated OOP implementation in R is the reference class implementation (RC). It is very similar to what you see in Python.

Creating an S3 Object

- It is very simple to define a new S3 class: simply add a character vector to the attribute of the object!
- There are no constructors required, and there are no checks to see whether it is a valid member of the class.
- Most of the objects you will encounter will be S3 objects.

```
foo1 <- list(X=rnorm(10),msg="hello")
class(foo1) <- "fooS3"
```

Creating an S3 Object

creating specific methods

```r
summary.fooS3 <- function(x) {
  str1 <- paste("Object has", length(x$X),
                "normal random variates.\n")
  cat(str1)
  str2 <- paste("Object message is: ", x$msg)
  cat(str2)
}
summary(foo1)
```

```
Object has 10 normal random variates.
Object message is:  hello
```

```r
plot.fooS3 <- function(x, ...) {
  hist(x$X, ...)
}
```

Interrogating S3 Objects

- As you would have observed, the object of class `fooS3` is just a list, and we can inspect it's elements as such.
- Other useful functions for learning about the object are:

```r
str(foo1)
```

```
List of 2
 $ X  : num [1:10] -0.557 0.588 -0.848 -0.357 -0.459 ...
 $ msg: chr "hello"
 - attr(*, "class")= chr "fooS3"
```

```r
methods(class="fooS3")
```

```
[1] plot    summary
```

Creating an S4 Object

- Creating an S4 object requires a call to the `setClass()` function.
- As part of this call, we have to define the *slots* that the class has. These are the attributes or data values of the class.
  - The data in slots can be accessed with the `object@slot` notation, or with the `slot(object, "slot")` function call.
- This call returns a default initialiser for the class.
- Following this, we extend generics to this class using the `setMethod()` function.

```r
fooS4 <- setClass("fooS4", slots=c(X="numeric",
                                 msg="character"))
foo2 <- fooS4(X=rnorm(20), msg="test")
```

Creating an S4 Object

```r
setMethod("summary", signature(object = "fooS4"),
  definition=function(object, ...) {
    str1 <- paste("Object has", length(object@X),
                  "normal random variates.\n")
    cat(str1)
    str2 <- paste("Object message is: ", object@msg)
    cat(str2)
  })

setMethod("plot", signature(x = "fooS4"),
  definition=function(x, y, ...) {
    hist(x@X, ...)
  })
```

Interrogating S4 Objects

In addition to the `methods()` function, we can also use the `showMethods()` and `slotNames()` functions to obtain more information if we know we are dealing with an S4 object.

Transactions

- The R package `arules` implements functions for working with transactional data.
- A **transaction** is a record of items purchased by a particular customer.
- The package implements a few new S4 classes. Let us study just one of them for now: `transactions`.

```
library(arules)    # install this first
trans_all <- read.transactions("../data/all_trans.txt",
                               format="single", header=TRUE,
                               cols=c("InvoiceNo", "desc2"),
                               sep=",")
```

Interrogating S4 Objects

transactions

```
isS4(trans_all)
```

```
[1] TRUE
```

```
slotNames(trans_all)
```

```
[1] "data"        "itemInfo"    "itemsetInfo"
```

```
str(trans_all, max.level = 2)
```

```
Formal class 'transactions' [package "arules"] with 3 slots
  ..@ data       :Formal class 'ngCMatrix' [package "Matrix"] with 5 slots
  ..@ itemInfo   :'data.frame': 38 obs. of  1 variable:
  ..@ itemsetInfo:'data.frame': 18270 obs. of  1 variable:
```

Transactions S4 Object

- The authors came up with their own class to represent a set of transactions.
- The set of all items is stored in the `itemInfo` slot. There were 38 of them.
- The invoice numbers are stored in the `itemsetInfo` slot. There were 18270 of them.
- The items bought within each transaction are stored as a sparse matrix in the `data` slot. Here is an example of the items purchased in the first few transactions:

```r
inspect(trans_all[1:3,])
```

```
    items                              transactionID
[1] {CANDLES,LIGHTS,OTHERS,UTENSILS}   536365
[2] {HAND WARMER}                      536366
[3] {DOOR STUFF,OTHERS,TOYS,UTENSILS} 536367
```

Other Methods for Transactions Class

- The reason the previous function worked is that the authors had
  - written a new function `inspect( )` that works on `transactions` class objects.
  - written a specific function '[' that works on `transactions`.
- What other methods are available for objects of class `transactions`?

```
methods(class="transactions")
```

- In order to know more about them, it is important to read through the documentation prepared by the authors:

```
vignette("arules")
```

- For our class, we will not be tested on writing S4 methods or classes. But it is important to know
  - how to manipulate them,
  - how to use the functions that have been written for them, and
  - how to extract the information you need from them.

Creating a RC Object

- To create a reference class object, we use the `setRefClass` function.
- RC objects contain a set of fields, which are analogous to the slots in S4 classes.
- We can also define the methods for this class with this same function.
- Unlike S3 and S4 classes, methods now belong to the class.
- Like Python, objects are passed by reference; they are not copied when assigned to a new symbol.

Creating an RC Object

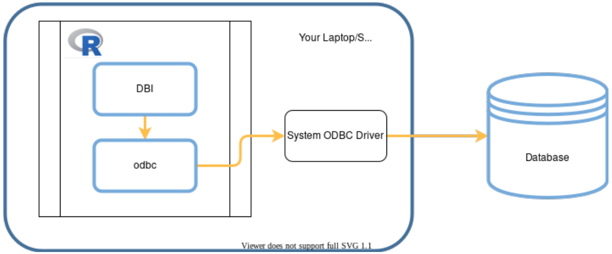
```r
fooRC <- setRefClass("fooRC",
  fields=list(X="numeric", msg="character"),
  methods=list(
    summary = function() {
     str1 <- paste("Object has", length(X),
                   "normal random variates.\n")
     cat(str1)
     str2 <- paste("Object message is: ",  msg)
     cat(str2)
    },
    plot = function(...) {
     hist(X, ...)
    }
 ))

foo3 <- fooRC$new(X = rnorm(10), msg="goodbye")
foo3$plot()
```

Summary

- There are many other concepts (chief among them being inheritance) that we have not discussed.
- The purpose was not to become experts in the OOP systems in R; that is appropriate for a more advanced course in R.
- The purpose of introducing the different types of OOP systems in R, was to make you aware of them.
- When you leave this class, you might have to deal with S4 objects (RC objects are very rare), and so it is important that you understand how they work.
- In our exam, we will only be tested on creating simple S3 objects.

Overview



1. Install the requisite packages on R.
2. Authenticate with the database server.
3. Query/Extract the data you need to work on. Some knowledge of the query language the database uses is necessary at this point.
4. Analyse the data.
5. Close the connection.

Mongo DataBase

- We are about to connect to a Mongo DataBase.
- Mongo is a NoSQL database; the data is not stored in a tabular format.
- Mongo allows us to store and retrieve *unstructured* data.
- Full documentation about Mongo can be found at this URL:
  - `https://www.mongodb.com/docs/v5.0/tutorial/getting-started/`

MongoDB®

Authenticating and Connecting

- Before connecting, ensure that
  1. You have installed the R package `mongolite`.
  2. You have placed the `mongo_user_pwd.txt` (from canvas) in your `src` folder.
  3. You are on campus, or have started a VPN connection to NUS
     - ★ The database is available at all times, but only from campus or through VPN.

```r
library(mongolite)
library(jsonlite)

credentials <- paste0(readLines("mongo_user_pwd.txt"),
                      collapse = ":")
connection_string <- paste0("mongodb://", credentials,
                            "@rshiny.nus.edu.sg:2717/test")
con2 <- mongo(verbose=TRUE, collection = "restaurants",
              url=connection_string)
con2$count()
```

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

Common Methods

- The code above connects to the `restaurants` collection on a `test` database.
- A collection is similar to a table in SQL databases.
- `con2` is an object of class "mongo". It is RC style object.

Common Methods for `mongo` Object

`con2$count()`: returns the number of records in the collection.

`con2$find()`: queries the collection and returns the matching records to R. More on this soon..

`con2$export()`: queries the collection and writes the matching records to a specified file.

`con2$insert()`: inserts records into the collection.

Collection Schema

- Here is what a single record in the collection looks like:

```
##  $ address      :List of 4
##   ..$ building: chr "1007"
##   ..$ coord   :List of 2
##   ..$ street  : chr "Morris Park Ave"
##   ..$ zipcode : chr "10462"
##  $ borough      : chr "Bronx"
##  $ cuisine      : chr "Bakery"
##  $ grades       :List of 5
##   ..$ :List of 3
##   ..$ :List of 3
##   ..$ :List of 3
##   ..$ :List of 3
##   ..$ :List of 3
##  $ name         : chr "Morris Park Bake Shop"
##  $ restaurant_id: chr "30075445"
```

Simple Queries in Mongo

Mongo DB queries are written as JSON strings:

- To return all Wendy's restaurants:

```
q1 <- toJSON(list(name = "Wendy'S"), auto_unbox = TRUE)
q1
```

```
## {"name":"Wendy'S"}
```

  - Equivalent SQL:

```
SELECT * FROM restaurants WHERE name = "Wendy'S";
```

Simple Queries in Mongo

cont'd

- To return all Wendy's restaurants in Manhattan:

```
q2 <- toJSON(list(name = "Wendy'S", borough="Manhattan"),
              auto_unbox = TRUE)
q2
```

```
## {"name":"Wendy'S","borough":"Manhattan"}
```

  - Equivalent SQL:

```
SELECT * FROM restaurants WHERE
    name = "Wendy'S" AND borough = "Manhattan";
```

Simple Queries in Mongo

cont'd

- To return all Wendy's restaurants in Manhattan or Brooklyn:

```
q3 <- toJSON(list(name = "Wendy'S",
                  borough=list(`$in`=c("Manhattan", "Brooklyn"))),
             auto_unbox = TRUE)
q3
```

```
## {"name":"Wendy'S","borough":{"$in":["Manhattan","Brooklyn"]}}
```

  - Equivalent SQL:

```
SELECT * FROM restaurants WHERE
   name = "Wendy'S" AND
   borough IN ("Manhattan", "Brooklyn");
```

- More help on Mongo queries can be found here:
  - https://www.mongodb.com/docs/v5.0/tutorial/query-documents/
  - https://www.mongodb.com/docs/v5.0/reference/sql-comparison/

Paginated Queries

- For queries that return a large number of records, it is common practice, when working with databases, to iterate over the returned records.
- Here is an example. The query retrieves a 10% sample of Wendy's restaurants.

```
it <- con2$iterate(query='{"name": "Wendy\'S"}',
        fields='{"_id": 0, "grades":1,  "address": 1}')
sampleA <- NULL
repeat{
  x <- try(it$batch(10))
  if(inherits(x, "try-error")) {
    message("Exiting cleanly..")
    break
  }

  sampleA <- c(sampleA, sample(x, size=1))
}
```

Salient Points

- When working with others, keep your passwords/tokens secret, but the same code should be usable by all.
- When working with databases, it is inevitable that you'll have to pick up the query language.
- Remember the constraints when creating JSON strings.
- RC-style objects are similar to those you encounter in Python and C++.
  - More help on mongolite here: `https://jeroen.github.io/mongolite/`
- When working with databases, it may take a long time to get your data. Learn how it is indexed to optimise your querying.
- Remember to close your connection when done with it:

```
rm(con2)
```

data.gov.sg

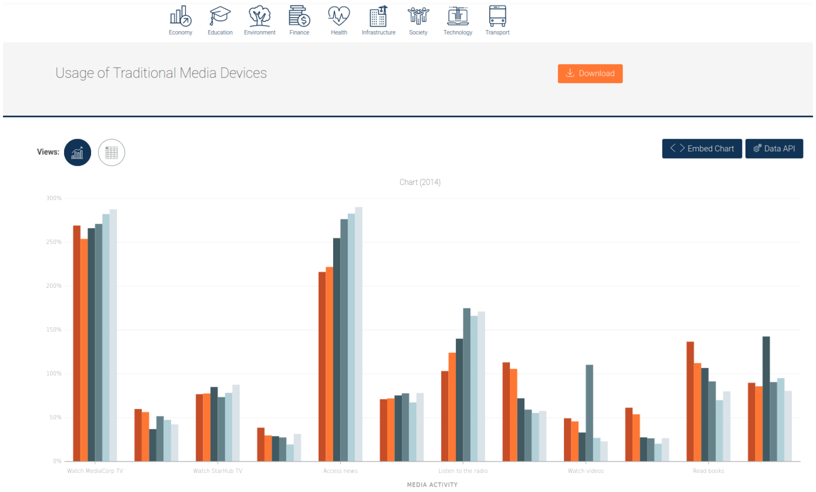
- `https://data.gov.sg/` was launched in 2011, as a one-stop portal for datasets made public by 70 government agencies.
- It contains datasets in the field of economy, education, environment, finance, health, infrastructure, society, technology and transport.
- From the website, datasets can be downloaded in csv or shp format.
- It is also possible to download the data using a script. The data would then be returned as a json object.

Usage of Traditional Media Devices

- Every year, the Infocomm Media Development Authority (IMDA) commissions a Media Consumer Experience Study.
- It aims to study consumer feedback on various aspects of the media content and services across broadcast, print and online platforms.
- The dataset in question is described as showing the *percentage of consumers who have ever used a traditional media device, e.g. TV, newspaper, etc., for media activities*.

Usage of Traditional Media Devices

https://data.gov.sg/dataset/usage-of-traditional-media-devices?resource_id=50c2820a-a18f-4514-9bb0-ff43048ddff5



Dowloading IMDA Data

- To download a file from a link, we can use the `download.file()` function in R.
- However, we need to know the URL of the file.

```
imda_file_url <-
"https://data.gov.sg/dataset/eab22bb4-4bb9-478d-bad3-b5293733337d/download"
return_val <- download.file(imda_file_url, "../data/imda.zip")
```

```
trying URL 'https://data.gov.sg/dataset/eab22bb4-4bb9-478d-bad3-b5293733337d/download'
Content type 'application/zip' length 10544 bytes (10 KB)
==================================================
downloaded 10 KB
```

- To read in the file, we open a connection to the csv file within the zip file.

```
con <- unz("../data/imda.zip",
           "usage-of-traditional-devices-for-media-activities.csv")
media_data <- read.csv(con, header=TRUE)
```

Downloading IMDA Data

cont'd

- The following checks the number of rows we have.

```
dim(media_data)
```

```
[1] 210   6
```

```
head(media_data)
```

```
  year   age     media_activity sample_size ever_used
1 2013 15-19 Watch MediaCorp TV         161      97.1
2 2013 15-19   Watch Singtel TV         161      32.9
3 2013 15-19   Watch StarHub TV         161      39.5
4 2013 15-19    Watch animation         161      30.9
5 2013 15-19        Access news         161      87.4
6 2013 15-19     Read magazines         161      46.7
```

Plotting IMDA Data

data for 20 – 29 age group

- Let us make a bar chart for the 20 – 29 year old age group.
- The following code will become comprehensible after topic 03; for now, I only need you to understand its purpose. It
  - filters out the groups we do not want,
  - converts the percentage to numeric (`ever_used` is a character vector now), and
  - arranges the dataset by decreasing percentage.

```
library(tidyverse)
young_adults <- filter(media_data, age == "20-29",
                        year==2015) %>%
  mutate(pct = as.numeric(ever_used)) %>%
  arrange(desc(pct))
```

Plotting IMDA Data

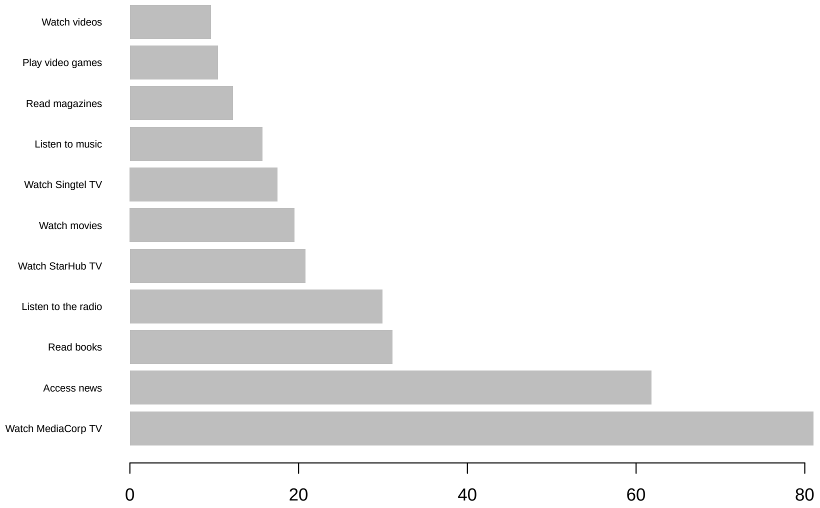
cont'd

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

```r
opar <- par(mar=c(5.1, 5.8, 2.1, 1.1))
barplot(young_adults$pct,
        names.arg = young_adults$media_activity,
        horiz = TRUE, cex.names = 0.6, las=1,
        border=NA)
par(opar)
```

Plotting IMDA Data

cont'd



Data.gov Developer API

- The following URL lists the real-time data that can be obtained from data.gov.sg:
  - `https://data.gov.sg/developer`
- These datasets are available through an API.
- An API on a server "listens" for requests, which come in the form of HTTP requests.
- HTTP requests are simply URLs, with a special form. The object returned is usually a JSON string.

Taxi Availability

- The following URL describes the endpoint for taxi availability in Singapore at a given time.
  - https://data.gov.sg/dataset/taxi-availability
- To retrieve the locations of available taxis at a particular time, use a GET request:

```
library(httr)
base_url <- "https://api.data.gov.sg/v1/transport/taxi-availability'
taxi_avail <- GET(base_url,
                  query=list(date_time="2022-08-01T09:00:00"))
```

Taxi Availability

cont'd

- It takes a little bit of investigation to extract the data we need from the returned object.
- The following code extracts the coordinates (in lat/long) of all available taxis at that point in time:

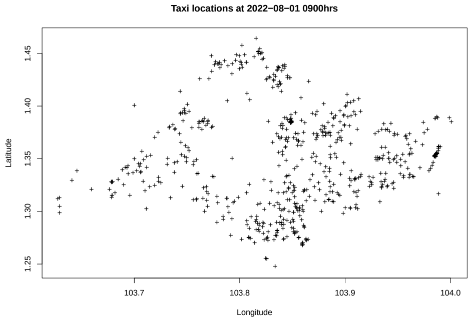
```r
taxi_data <- content(taxi_avail)
x_coords <- sapply(taxi_data$features[[1]]$geometry$coordinates,
                   `[[`, 1)
y_coords <- sapply(taxi_data$features[[1]]$geometry$coordinates,
                   `[[`, 2)
```

Taxi Availability

cont'd

- Here's a basic visualisation of the data:

```
plot(x_coords, y_coords,
     main="Taxi locations at 2022-08-01 0900hrs",
     asp=1, pch="+",
     xlab="Longitude", ylab="Latitude")
```



`rvest` Package

- The `rvest` package in R provides convenient routines for scraping web pages in R.
- The main functions that you will need are:

  `read_html()`: This will read in a web-page and store it in R.

  `html_nodes()`: This will extract nodes from a web-page in R. Nodes are specified by a *pattern* that they contain.

  `html_text()`: This will extract the text from a node.

  `html_table()`: If there is a table in a node, this will extract it.

  `html_structure()`: This function allows you to inspect the structure of a node.

Structure of HTML pages

HTML is a markup language. It contains HTML tags and text in a **tree-like** structure. At each **node** of the tree, the tags contain **properties** and *name=value* **attributes** that specify how the **text** should be displayed.

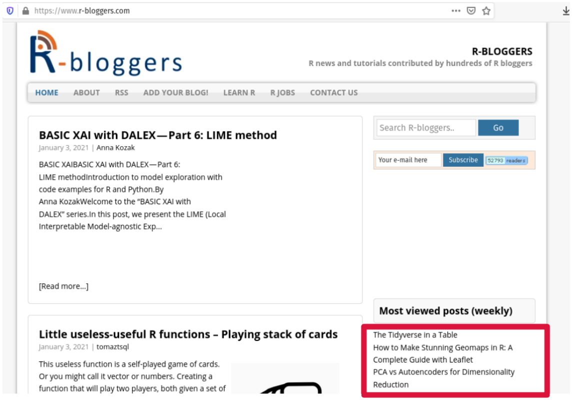
Identifying Elements in a Web-Page

- The general procedure for extracting text from a HTML page is to do the following:
  1. Read in the HTML page as a html document within R (this is a custom class).
  2. Extract the section (or nodes).
  3. Extract the text from the nodes.
- In order to select the nodes, you typically have to **inspect** the web-page in order to identify where the information you need is stored.
- Chrome, Firefox and Safari provide functionality to identify this information.

Selector Gadget

- Content is stored within tags, that could be identified by
  - a class name
  - a type
  - an id, and so on.
- This identifying information has to be passed on to the `html_nodes()` function.
- The selector gadget at `https://selectorgadget.com/` provides a tool that returns the appropriate text string to use.
- Try a few of the exercises at `https://flukeout.github.io/` to get a better idea of how HTML elements can be selected.

R-bloggers Website



https://www.r-bloggers.com

R-BLOGGERS

R news and tutorials contributed by hundreds of R bloggers

BASIC XAI with DALEX—Part 6: LIME method

January 3, 2021 | Anna Kozak

BASIC XAIBASIC XAI with DALEX—Part 6:
LIME methodIntroduction to model exploration with
code examples for R and Python.By
Anna KozakWelcome to the "BASIC XAI with
DALEX" series.In this post, we present the LIME (Local
Interpretable Model-agnostic Exp...

[Read more...]

Little useless-useful R functions – Playing stack of cards

January 3, 2021 | tomaztsql

This useless function is a self-played game of cards.
Or you might call it vector or numbers. Creating a
function that will play two players, both given a set of

Most viewed posts (weekly)

The Tidyverse in a Table
How to Make Stunning Geomaps in R: A Complete Guide with Leaflet
PCA vs Autoencoders for Dimensionality Reduction

- The r-bloggers website at https://www.r-bloggers.com/ contains links to articles on R.
- If you click on a link in the "Most viewed posts" section, you will be taken to the corresponding r-bloggers article.
- Suppose we wish to obtain the **link address** and the **title** of each of these most viewed posts, using R.

R-bloggers Website

read and extract nodes

- Our first task is to read in the entire page.

```r
library(rvest)
library(xml2)
rbloggers_page <- read_html("https://www.r-bloggers.com/")
```

- After using the selector gadget, we find that the appropriate path to the links in the most-viewed section is as follows:

```r
nodes <- html_nodes(rbloggers_page, "#wppp-3 a")
```

- The `nodes` object contains a list. If we print it, we see something similar to this:

```r
nodes
```

```
{xml_nodeset (7)}
[1] <a href="https://www.r-bloggers.com/2020/12/the-tid...
[2] <a href="https://www.r-bloggers.com/2020/12/how-to-...
[3] <a href="https://www.r-bloggers.com/2018/07/pca-vs-...
```

R-bloggers Website

inspect nodes

- The `nodes` object can be accessed like a list. If we inspect its structure, we see that it is quite simple:

```
html_structure(nodes[1:2])
```

```
[[1]]
<a [href, title]>
  {text}

[[2]]
<a [href, title]>
  {text}
```

- The link is stored in the `href` attribute, and the the title in the `title` attribute.

R-bloggers Website

extract text

- To extract what we need, we can use the following code:

```
links <- html_attr(nodes, name="href")
titles <- html_attr(nodes, name = "title")
```

- To extract all attributes at one go, we can use

```
links_and_titles <- html_attrs(nodes)
```

- Notice there is no need for a "`for`" loop or even any `apply` function.

Further Resources

- Sometimes, you will need to fill in a form (usually a search box) before the web-page with the data you wish to scrape is generated.
- `rvest` provides basic functionality for this, but it does not always work.
- Other times, the website you wish to scrape will forbid bots from submitting forms. An example of this is Property Guru Singapore.
- In those situations, you will need to mimic a true browser; `rvest` is not sufficient for this.
- You may want to look up the following alternatives that have been helpful to me in the past:
    1. `Selenium`, a Python package, allows you to click buttons, submit forms, etc.
    2. If you use a Mac, the Applescript program allows you to do the same.
- *Selenium and Applescript will not be tested in our exams.*

Summary

- Importing data becomes complicated when data is not stored in a friendly format.
- If there is a mixture of binary and text characters, it can be challenging.
- When reading data from the web, we need to have some creativity to identify patterns or keywords that we can then use in a loop.
- It is unlikely to be the same every new dataset you need, but the experience you gather will help you along.