Data Importing

Files	
Notes	
Status	
Topics	Week 2

Lecture Notes

https://s3-us-west-2.amazonaws.com/secure.notion-static.com/8d6a0ba6-bebd-4717-8148-5a78a7be708a/02_data_import_(Annotated).pdf

---

CSV Files

CSV files are Comma-Separated Values whereby each of the values are being separated by `,` commas

- **Structure**
  - Text files
  - Contains an optional header, listing the column names
  - Observations separated by commas within each row
- **Help**

  `?read.csv` - Help page for csv
- **Note**
  - Excel worksheets can be exported to csv format easily
- **General Procedure**
  - Before reading in the document
    1. View the csv file in a text editor
    2. Read through the first few lines first
    3. Check the amount of metadata, the presence/absence of headers, and the number of columns
    4. 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 in the document

    Use `str()` to check the following
    1. Check if the correct number rows are read in
    2. Check if the column names and column classes are correctly assigned
    3. Check if missing values are read in correctly
- **Functions**
  - **Reading in CSV Files**

    `read.csv()` - Reads in the csv file for `sep=","`
    - **Arguments**

      `file` : File path to the csv file

      `header` : `TRUE` / `FALSE` , Absence/ presence of the header row

Default: `TRUE` therefore, the first row is read in as a header

`skip` : Number of comment lines at the beginning, it will skip the number of lines that is stated and start reading in from the line after the skipped lines

`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

Default: `TRUE` which means that the string columns will become factors

Set to `FALSE` if we want to keep them as strings

`col.names` : The names used to identify the columns in the table

`colClasses` : Used to specify the classes for each of the columns that are being read in

We just need to pass in a vector that is of the same length as the number of columns and specifying the data type for each of the columns

Note: If we specify `NULL` for the data type, it will not read in that column into the data frame

`read_csv()` - Reads in the csv file but as a `tibble`

Arguments

`file` : File path to the csv file

`header` : `TRUE` / `FALSE` , Absence/ presence of the header row

`skip` : Number of comment lines at the beginning, it will skip the number of lines that is stated and start reading in from the line after the skipped lines

`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

`col.names` : The names used to identify the columns in the table

`colClasses` : Used to specify the classes for each of the columns that are being read in

We just need to pass in a vector that is of the same length as the number of columns and specifying the data type for each of the columns

Note: If we specify `NULL` for the data type, it will not read in that column into the data frame

`read.delim()` - Reads in a tab delimited file where each of the values are separated by `"\t"`

Arguments

`file` : File path to the tab delimited file

`header` : `TRUE` / `FALSE` , Absence/ presence of the header row

Default: `TRUE` therefore, the first row is read in as a header

`skip` : Number of comment lines at the beginning, it will skip the number of lines that is stated and start reading in from the line after the skipped lines

`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

Default: `TRUE` which means that the string columns will become factors

Set to `FALSE` if we want to keep them as strings

`col.names` : The names used to identify the columns in the table

`colClasses` : Used to specify the classes for each of the columns that are being read in

We just need to pass in a vector that is of the same length as the number of columns and specifying the data type for each of the columns

Note: If we specify `NULL` for the data type, it will not read in that column into the data frame

`read.table()` - Read any tabular file as a data frame

- This is especially useful if we have files that have weird delimiters

Arguments

`file` : File path to the file

`sep` : Specifies the separator that is used for the file

E.g. `"/" "\t" "\n"`

`header` : `TRUE` / `FALSE` , Absence/ presence of the header row

Default: `FALSE` therefore, the first row is not read in as a header

`skip` : Number of comment lines at the beginning, it will skip the number of lines that is stated and start reading in from the line after the skipped lines

`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

Default: `TRUE` which means that the string columns will become factors

Set to `FALSE` if we want to keep them as strings

`col.names` : The names used to identify the columns in the table

`colClasses` : Used to specify the classes for each of the columns that are being read in

We just need to pass in a vector that is of the same length as the number of columns and specifying the data type for each of the columns

Note: If we specify `NULL` for the data type, it will not read in that column into the data frame

▼ **Data Checks**

`unique()` - Check for the unique values in the given vector

`is.na()` - Returns a logical vector on whether there are any empty values in the data

```
apply(heights , 2, function (x) sum(is.na(x))) # one of the ways to cehck the number of empty values
```

▼ **Plotting**

`hist()` - Plots a histogram (Look under *Basic R* → *Plotting*)

▼ **Unable to use `read_csv()` to read in properly**

`readLines(filename)` - Reads in the file line by line which allows us to parse it afterwards

Excel Files

▼ **Packages**

`readxl` - Package that allows for reading of data from `xls` and `xlsx`

- 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.

`gdata` - Package that allows for data manipulation

- Elegant extension of `utils` package

- Entire suite of tools for data manipulation
- Supercharges basic R
- Support for XLS format and also XLSX (but this requires additional driver)

This is the way gdata reads in the data

▼ Functions

▼ Loading of Package

`library(readxl)` - Package for reading Excel Documents

`library(tibble)` - Package for `tibble` which is a subclass of data frames

`library(gdata)` - Package for reading Excel Documents

▼ Reading of Data

`read_excel()` - Reads in the excel file stated

Arguments:

`file` - Filename of the excel file

`col_names` - Logical values. Similar to `headers` in `read.csv()` where we state where there are column names

- It can be `TRUE` whereby the first row will be read in as a header
- `FALSE` and R will decide the row names by itself and reads in the first row as data
- We can also specify our own column vectors by passing in a character vector

`col_types` - Specify what are the types of the data that should be contained in each of the columns

- `NULL` default setting and excel will just read and it will guess the type of the data
- `"blank"` - We can pass this in for the column data type if we do not want to include the column in our data
- We can also specify the type of the data that we want by passing in the data types as a vector

`skip` - States the number of rows to be skipped before starting to read in data

`n_max` - States the maximum number of rows to look up for non-empty cells

`range` - States the range of the data that we will read in. We can use the excel notation of the cells (E.g. `"C7:E9"`) and for the empty cells within it will be filled in with `NA`

`sheet` - States the sheet to be read in. Can either be the name of the sheet or the position of the sheet

Note:

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

1. `read_excel()` will use the smallest rectangle that contains the non-empty cells by discovered it itself
2. Bounding the rows using `skip` and `n_max`
3. Specifying the specific range using `range`

`read.xls()` - `gdata` version of reading in data for excel

`sheet` - States the sheet to be read in. Can either be the name of the sheet or the position of the sheet

`file_path` - State the file path of the xls file that we want to read in

▼ More Information

`vignette('sheet-geometry')`

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

▼ Finding out the name of Sheets in a file

`excel_sheets()` - Provides a character vector of the sheets within the excel file

▼ Reading in Multiple Files at the same time

`lapply(excel_sheets("filepath"), read_excel, path="filepath")`

- We are applying the argument of the sheets through the `read_excel` function for all the different sheets that are generated through the `excel_sheets("filename")`
- We specify `path="filename"` so that the argument is always supplied to the `read_excel` function and the only argument left is the sheets

---

JSON Files

- It is a text format for storing structured data, it does not have to be a file, it can just be a string of code
- Note that we may have to convert our data after reading it in and change it into other forms

▼ Packages

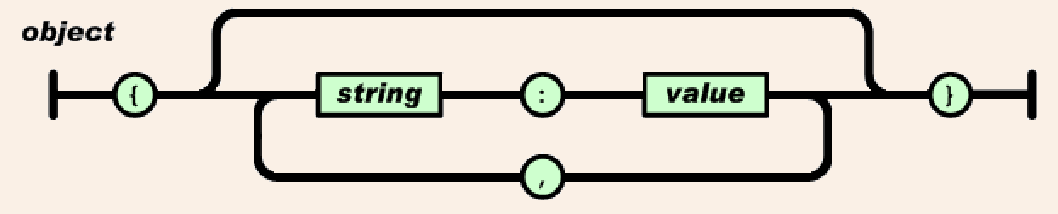
`jsonlite` - Package for us to parse and generate JSON files

▼ Structure

It has mainly 2 structures

▼ Object

- Unordered collection of name/value pairs

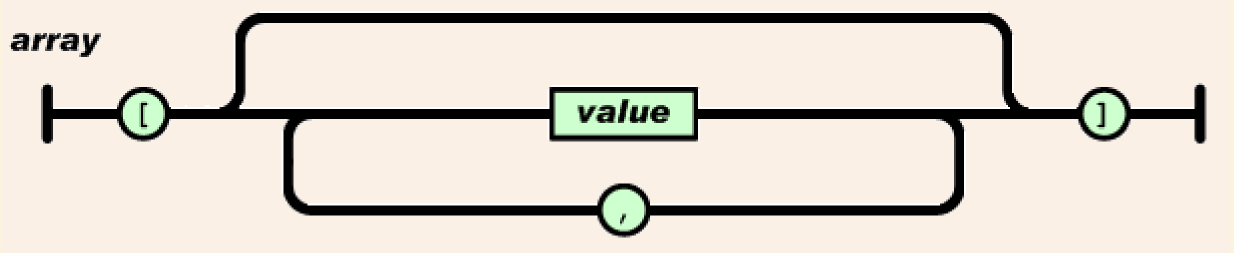


Visualisation of how the object structure is

- Contained within `{ }`
- Name is followed by `:`
- Each pair is separated by `,`

▼ Array

- Ordered list of values



Visualisation of how the object structure is

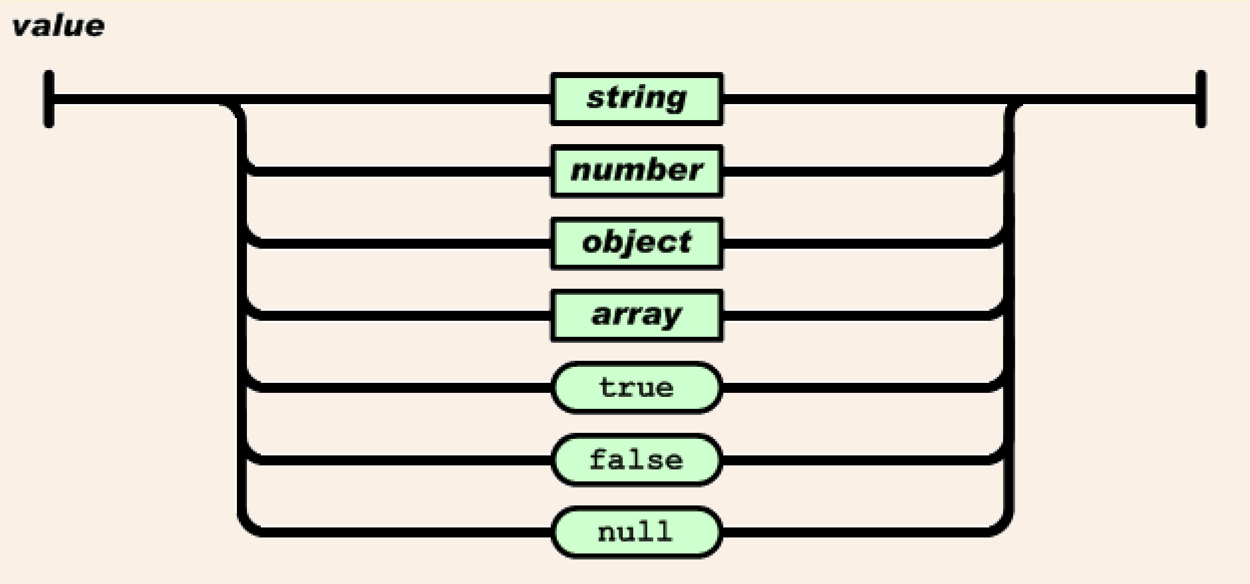
- Contained within `[ ]`
- Each pair is separated by `,`

▼ Values

- String

  Note that for strings that must be in double quotations and single quotations does not work
- Number
- Object
- Array

- true
- false
- null



▼ Functions

▼ Reading JSON Objects

`fromJSON(txt)`

- This function reads in the JSON object either from files, the web, or even straight from the console
- We just have to pass the object through the function and it will automatically parse it for us and cast them in with the way they are homogenous
- Note that we can only use `fromJSON` on 1 object or 1 array at a time
- If we read in an object, it will be stored as a list with each of the pairs being a vector under this list

`toJSON(object, pretty)`

- Converts the R object into a JSON string
- `pretty` - Setting this to `TRUE` will prettify the string

`readLines(txt)`

- This function reads in each of the lines and stores them as a character vector
- This allows us to read in multiple lines at one time

```
# We can just make use of lapply and apply fromJSON to each of the lines
json_list <- lapply(all_lines, fromJSON)
```

`prettify(string)`

- This is for us to visualise the JSON string if we read it in using `readLines`

`minify(string)`

- Makes the JSON string as concise as possible

▼ ndjson Format

`?stream_in` - Help page for the function stream_in

`stream_in()` - Allows for multiple lines of JSON objects to be read in at the same time and stores them into a `data.frame`

- We just need to supply a connection to a JSON object, we can make use of `file(file_url)` to open up a connection to a file that we have on our computer and it also helps us to close the file as well

Object Oriented Programming (OOP)

Note

- Methods for S3 & S4 objects do not belong to the class. We will define *generic functions* instead and we will write specific methods for each class and R will dispatch the appropriate method for the classes when the *generic function* is called

S3 Objects

- Most common and easiest to create. However, there are not much checks imposed on these classes

Functions

Creating an S3 Object

- Note that no constructors are required, and no checks to see whether it is a valid member of the class

```
class(x) <- "new_class"
```

- We just have to use the `class()` function which will act like a setter

Creating Specific Methods

- For methods, we will just have to define it under the generic function that we want to. R will dispatch the correct method if the object has been passed as an argument

```
summary.fooS3 <- function(x){
  x <- x+1
  x
}
```

- We can just call the generic functions as it is without adding the `.class_name` and passing in the class object

Interrogating S3 Objects

`str(object)` - Checks the structure of the object and also tells us about the attribute of the object (which also includes the class)

`methods(class="class_name")` - Lists out the methods that are available for the class

S4 Objects

- This is less common as compared to S3 objects. The creation is much more rigorous as compared to S3

Functions

Creating an S4 Object

`setClass()` - We will need to call this function to create an S4 object

- Note that we will have to set the `slots` as well for the object which are the arguments that we need to pass it in
- This function will create a default initialiser so that we can use to create the objects of that class

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

Creating Specific Methods

- Same thing as for S3 objects, we will extend the generic functions and R will dispatch the appropriate methods for the classes

`setMethod()` - This will set the method for the class

```
setMethod("name_of_generic_function", signature(object = "class_name"), definition=function(object, ...){
})
```

- We will need to pass in the name of the generic function as the first argument as a string
- Then we pass in the `signature(object = "class_name")` where we will state which is the class that we want to assign the method to
- `definition` will be the part whereby the specify what the function does and what arguments it takes in. `...` is added so that we can accept the additional arguments as per the generic function

▼ Interrogating S4 Objects

`str(object)` - Checks the structure of the object and also tells us about the attribute of the object (which also includes the class)

`methods(class="class_name")` - Lists out the methods that are available for the class

`showMethods(generic_function)` - Lists out the S4 methods that are under this generic function if there are any. It will show which are the objects that are using this function

`isS4(object)` - Checks whether the object is an S4 object

`slotNames(object)` - Checks what are the slot names for the object

`slot(object, "slotname")` - This returns the value for the slot under the slotname of the given object

`object@slot` - By using the `@` symbol followed by the slotname, it acts like a getter but using `slot()` is the preferred way

▼ Reference Class (RC) Objects

- This is the least common amongst the 3 Classes
- Methods now belong to the class for RC objects
- Contains a set of fields which are similar to slots in S4 class
- Objects are passed by reference, they are not copied when assigned to a new symbol

▼ Functions

▼ Creating a Reference Class

`setRefClas()` - Allows us to create a reference class object and it states the `fields` and also the `methods` that are within the function

```
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, ...)
}
))
```

- First argument is the name of the class that we want to create
- Second argument is the `fields` which are the parameters that we require to create the object and we will state the object type of each of the parameters
- Third argument is the `methods` whereby we will state the methods for the class and each of the methods are listed with the function definitions being created within this

▼ Creating a new RC Object

`class$new()` - We can make use of the `$new` method and pass in the relevant arguments required to create the object

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

▼ **Calling Methods**

`$method` - We can make use of the `$method` to directly call the method from the object itself

```
foo3$plot()
```

Spatial Data

- It is used for us to define objects using geometry

▼ **Packages**

`sp` - Oldest spatial data package

`sf` - Simple features package. Recent package that uses simpler, native data structures to represent spatial objects as simple features

- Functions and methods that operate on simple features are prefixed with `st_`

`tidyverse`

`rgdal` - Library to obtain EPSG codes

`leaflet` - Helps us with the plotting of spatial objects on maps. This function creates a Leaflet map widget using htmlwidgets. The widget can be rendered on HTML pages generated from R Markdown, Shiny, or other applications.

▼ **File Formats**

Most of these file types can be read in using `sf`

1. ERSI shapefiles
2. GeoJSON files
3. KML Files

▼ **Simple Feature**

▼ **Features**

- It is a thing or an object in the real world
- A set of features can form another feature

▼ **Characteristics:**

- **Geometry**, describing where the feature is located
- **Attributes**, which describes other properties that the object may have

▼ **Types of Geometry**

▼ **Note:**

- All geometries are composed of points, which are coordinates in a 2-, 3- or 4-dimensional space.
- All points in a geometry have the same dimensionality. In addition to X and Y coordinates, there are two optional additional dimensions:
    - A Z coordinate, denoting altitude
    - An M coordinate, denoting some measure that is associated with the point. This is typically unused.

There are also 'MULTI' versions of the geometry types below and they store several features of one object

▼ **POINTS**

- A single point

- For things with insignificant dimensions

▼ **LINESTRING**

- A sequence of points
- Normally for Roads

▼ **POLYGON**

- A sequence of points forming a closed ring. There could be subsequent rings within this one

▼ **Coordinate Reference System (CRS)**

- It allows us to represent a bumpy ellipsoid (Earth) on the plane (paper/screen)

- ***Longitude*** - Horizontal Reference
- ***Latitude*** - Vertical Reference

▼ **Geographical/Spheroid**

- Represented in terms of degrees
- Associated with:
  - an ellipse a model of the shape of the earth,
  - a prime meridian definig the origin in longitude (easting), and
  - a datum, which anchors a specific geographical CRS to an origin point in 3 dimensions.
- **Example**: *WGS84 (Most Common)*

▼ **Projected**

- This includes the items in the spheroid CRS
- We project it onto a flat surface
- It also includes
  - A specific geometric model projecting to the plane, and
  - Measures of length
- **Example**: UTM Zones, or Web Mercators

▼ **Specifying a CRS**

- Two Different Ways to Represent a CRS

▼ ***proj4string***

- Older method
- Consists of tags and corresponding values with a specification of `tag=value` format

```
+ proj = tmerc + lat_0 =1.366666666666667
+ lon_0 =103.8333333333333 +k=1
+x_0 =28001.642 +y_0 =38744.572
+ ellps = WGS84 + units =m + no_defs
```

▼ EPSG

- Stands for European Survey Petroleum Group, they maintain a list of CRS used throughout the world

```
https://epsg.org/home.html
```

- Each of the CRS is represented with an integer ID and this allows for precise projection to be changed/updated in the background

▼ Example

- EPSG Code for SVY21 projection system is 3414
  - This is created by Singapore Land Authority for the projection of Singapore
  - The units will be in metres
- EPSG Code for WGS84 is 4326

▼ Functions

▼ Reading in of spatial objects

`st_read(file_path, drivers= )` - This allows for spatial objects to be read in and we just have to specify the path to the file and also the `drivers` which is the file type

```
sg_poly[1:10, c("Name", "geometry")]
```

```
Simple feature collection with 619 features and 1 field
geometry type:  POLYGON
dimension:      XYZ
bbox:           xmin: 103.5817 ymin: 1.158762 xmax: 104.4112 ymax: 1.513675
z_range:        zmin: 0 zmax: 0
geographic CRS: WGS 84
First 10 features:
     Name                       geometry
1   kml_1 POLYGON Z ((103.9564 1.3313...
2   kml_2 POLYGON Z ((103.9575 1.3187...
3   kml_3 POLYGON Z ((103.9607 1.3211...
4   kml_4 POLYGON Z ((103.9627 1.3208...
5   kml_5 POLYGON Z ((103.9628 1.3162...
6   kml_6 POLYGON Z ((103.964 1.32373...
7   kml_7 POLYGON Z ((103.9664 1.3260...
8   kml_8 POLYGON Z ((103.9689 1.3232...
9   kml_9 POLYGON Z ((103.9689 1.3252...
10 kml_10 POLYGON Z ((103.969 1.31795...
```

sf

When we read in a spatial object, we will have the rows on top first with the description on the geometry type etc. , make sure we have the CRS. Below that will be the data for our simple feature and it will contain a name and the stated number of fields after that

---

▼ Type of Objects under `sf`

`sf` : An object that inherits from a `data.frame`. It stores both the attributes and geometry of each object in a single row.

```
sg_poly[1:10, c("Name", "geometry")]
```

```
Simple feature collection with 619 features and 1 field
geometry type:  POLYGON
dimension:      XYZ
bbox:           xmin: 103.5817 ymin: 1.158762 xmax: 104.4112 ymax: 1.513675
z_range:        zmin: 0 zmax: 0
geographic CRS: WGS 84
First 10 features:
     Name                      geometry
1   kml_1 POLYGON Z ((103.9564 1.3313...
2   kml_2 POLYGON Z ((103.9575 1.3187...
3   kml_3 POLYGON Z ((103.9607 1.3211...
4   kml_4 POLYGON Z ((103.9627 1.3208...
5   kml_5 POLYGON Z ((103.9628 1.3162...
6   kml_6 POLYGON Z ((103.964 1.32373...
7   kml_7 POLYGON Z ((103.9664 1.3260...
8   kml_8 POLYGON Z ((103.9689 1.3232...
9   kml_9 POLYGON Z ((103.9689 1.3252...
10 kml_10 POLYGON Z ((103.969 1.31795...
```

sf

Bracketed area is the `sf`

As seen here the name column are the attributes that are within this spatial object

---

`sfc`: The column (within the object of class `sf`) of list objects that stores the geometries for all the simple features.

```
sg_poly[1:10, c("Name", "geometry")]
```

```
Simple feature collection with 619 features and 1 field
geometry type:  POLYGON
dimension:      XYZ
bbox:           xmin: 103.5817 ymin: 1.158762 xmax: 104.4112 ymax: 1.513675
z_range:        zmin: 0 zmax: 0
geographic CRS: WGS 84
First 10 features:
     Name                      geometry
1   kml_1 POLYGON Z ((103.9564 1.3313...
2   kml_2 POLYGON Z ((103.9575 1.3187...
3   kml_3 POLYGON Z ((103.9607 1.3211...
4   kml_4 POLYGON Z ((103.9627 1.3208...
5   kml_5 POLYGON Z ((103.9628 1.3162...
6   kml_6 POLYGON Z ((103.964 1.32373...
7   kml_7 POLYGON Z ((103.9664 1.3260...
8   kml_8 POLYGON Z ((103.9689 1.3232...
9   kml_9 POLYGON Z ((103.9689 1.3252...
10 kml_10 POLYGON Z ((103.969 1.31795...
```

sfc

Bracketed area is the `sfc`

The columns represents all the geometry feature that is related to each of the objects that are under this spatial object

---

`sfg`: The particular geometry of an individual simple feature.

```
sg_poly[1:10, c("Name", "geometry")]
```

```
Simple feature collection with 619 features and 1 field
geometry type:  POLYGON
dimension:      XYZ
bbox:           xmin: 103.5817 ymin: 1.158762 xmax: 104.4112 ymax: 1.513675
z_range:        zmin: 0 zmax: 0
geographic CRS: WGS 84
First 10 features:
     Name                      geometry
1   kml_1 POLYGON Z ((103.9564 1.3313...
2   kml_2 POLYGON Z ((103.9575 1.3187...
3   kml_3 POLYGON Z ((103.9607 1.3211...
4   kml_4 POLYGON Z ((103.9627 1.3208...
5   kml_5 POLYGON Z ((103.9628 1.3162...
6   kml_6 POLYGON Z ((103.964 1.32373...
7   kml_7 POLYGON Z ((103.9664 1.3260...
8   kml_8 POLYGON Z ((103.9689 1.3232...
9   kml_9 POLYGON Z ((103.9689 1.3252...
10 kml_10 POLYGON Z ((103.969 1.31795...
```

sfg

Bracketed area is the `sfg`
An `sfg` is a specific simple feature's geometry

▼ Getting the Geometry of Spatial Objects

`st_geometry()` - Get, set, or replace geometry from an `sf` object

We can pass in our `sf` object here and get the geometry values of it and we can pass it into our plot function and it will be in terms of x and y

▼ Plotting of Spatial Objects

`plot(object)`

- If we do not specify anything, it will try to plot for everything including all the attributes
- There will be colours assigned for the different attributes

---

`plot(st_geometry(object), border=grey() axes=TRUE)`

- This will plot only the outline of the object and only the geometry that is assigned to the object.
- `grey(intensity, transparency)`
  - We could specify how much intensity that we want for our borders and how transparent we want it to be so that our graph will not be so intense and we can overlay other plots on it
  - `intensity` - Range from 0 to 1, the closer it is to 0, the closer it is to black and the closer to 1, the closer to white it will be
  - `transparency` - Range from 0 to 1, it shows how much items can pass through it. If it is closer to 0, it is more transparent, if it is closer to 1, it is more opaque

    `0.25` - 4 objects before it blocks them out

    `0.10` - 10 objects before it blocks them out

`grid()`

- Plots a grid on the plotted graph

▼ Checking the CRS for the object

`st_crs(object)`

- Tells us what is the coordinate reference system that the object is using

▼ Changing the CRS for the object

`st_transform(object, crs = )`

- We can just use this and specify the `crs` which will be in the EPSG format and it will return a spatial object that has that stated `crs`

▼ **Changing of data into `sf` objects**

`st_as_sf(object, coords=(longitude, latitude)`

- Pass in the object and we will need to specify which are the columns that has the *longitude* and the *latitude* (x, y)
- Remember to add in the `crs` for the object after converting using `st_crs(object)<- crs_code`

▼ **Using `leaflet` to plot map widgets**

- We will need to use the `leaflet` library to produce map widgets for our plots which are much more aesthetically pleasing
- We will make use of the pipe operator `%>%` when we are trying to go from one step to the next

▼ **Things to note**

- Note that the `sf` object that we pass inside should have 2 dimensions only (i.e. XY dimension). If there are more dimensions, we can make use of `st_zm()` to reduce the dimensions
- Note that the CRS for the spatial object should be in ***WGS 84 (i.e. EPSG 4326)***

`leaflet(data,...)` - This function creates a Leaflet map widget using htmlwidgets. The widget can be rendered on HTML pages generated from R Markdown, Shiny, or other applications.

Note that we can pass in our spatial object that has points here

`addTiles(leaflet_object)` - Add the map tiles onto the map widget

`addCircles(leaflet_object)` - Add the points using the spatial objects that are indicated

`addProviderTiles(leaflet_object, provider$Stamen.Toner)` - Add map tiles from another provider instead of the default one

`addPolygons(leaflet_object, data = , col = , group = " ")` - This adds polygons which are shapes made up of many points. We will need to pass in a spatial object through `data` here and we can name the polygon here with `group` and it can be any name. We can make use of this name to reference to this polygon.

`addLayersControl(leaflet_object, overlayGroups = " ")` - This will create a button for us control the layers that are visible. We can pass in groups that we have labelled previously under `overlayGroups`

```
## code that plots out the planning area for tutorial 4 onto a real map
library(leaflet)
# https://rstudio.github.io/leaflet/
taxis2 <- st_transform(taxis, 4326)
leaflet(taxis2) %>% addTiles() %>% addCircles()
leaflet(taxis2) %>% addProviderTiles(providers$Stamen.Toner) %>%
  addCircles() %>%
  addPolygons(data=pln2, col="red", group="Planning area") %>%
  addLayersControl(overlayGroups="Planning area")
```

Data From the Web

▼ **Ways to get Data from the web**

1. **Download from some URL**

▼ **Note**

- We can make use of the Data API that is provided for the websites if they are provided and they will allow use to collect the data by make requests from it
- We can note the different ways that the request can be made for the data from the url and we could also what are the things we can specify to control the data that we take in. For the `data.gov.sg` data, we can make use of the `limit` and `offset` with the `&` sign to denote how much of the data that we want
- Check the structure of the object that is read in so that we know what we have and what we can play with

Data.gov.sg

Singapore's open data portal

https://data.gov.sg/

This is Singapore's website for data that is collected and we can make use of the API for it

Functions

`fromJSON()` - We can pass in the URL for the data here and it can be parsed as a JSON object as well

- `limit` - Note that when we make a request, the default number of matches that we will take in is 100 records. However, we can set the limit under the url that we are requesting from
- `offset` - Note that we make a request, we can choose the offset that we want which means it is the number of records that we skip before we start to take in the records

`download.file(url, dest_path)` - If we want to download any files from online

- `url` - URL of the document we want to download
- `dest_path` - Path on the computer that we want to download the file to, inclusive of the file name
- We can just pass in the url of the file that we want to download and it will do so

Functions for reading in documents

Can read in directly:

- `read_csv`
- `read_tsv`
- `read.csv`
- `read.xls`

Cannot read in directly and need to have local file instead:

- `read_excel`
- `load`

`file.path(path1, path2)` - Join paths together

`load(file)` - Used to load .RData files into our workspace

2. **Scraping from a website**

Package / Tools

`rvest` - Provides convenient routines for scraping web pages in R

selector gadget

`beautifulsoup4` (Python)

Usage

- Need to routinely get data from a site
- Too much to cut/paste from

General Note

Structure of HTML

- HTML is a markup language
- Contains 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

```
<tagtype1 class = " "
          id = " "
          href = " "> # Those inside are the attributes

  text could be written here
  <tagtype2> </tagtype2> # We could have nested nodes inside of another


</tagtype1>
```

▼ **Identification of Content**

▼ **Types of tags**

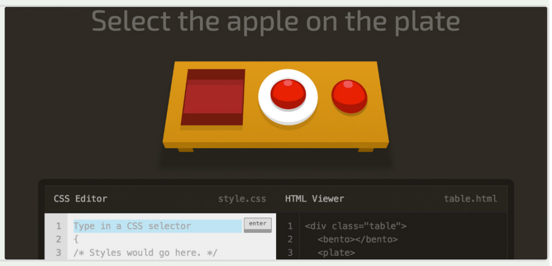
- Class Name
- Type
- Id

- We will need to pass these information to `html_nodes()` in order to get the relevant text out
- We can make use of the *Selector Gadget* in our webpages to figure out what is the tag that we should type in
- Practice for how HTML elements can be selected:

CSS Diner

You're about to learn CSS Selectors! Selectors are how you pick which element to apply styles to. Exhibit 1 - A CSS Rule p { margin-bottom: 12px; } Here, the "p" is the selector (selects all elements) and applies the margin-bottom style.

https://flukeout.github.io/



▼ **General Procedure for extracting text**

1. Read in the HTML page as a html document within R (this is a custom class)
2. Extract the section (or nodes)
   a. To understand which section/nodes we need to extract, we can use the ***inspect*** function in most web browsers now to analyze the structure of the HTML page
3. Extract the text from the nodes

▼ **Functions**

`library(rvest)` - Reading in of the library that we need

`read_html()` : Reads in a web-page and store it in R

This function accepts either a file, URL or a string that contains HTML information

We can just pass in the weblink for any website and it will be taken in as a HTML site

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

Example of a function call

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

We will supply the function with a html_object and also the tags (Which are the things that in the `< >` what we want to extract from then object

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

Example of a function call

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

We just need to input a node inside and it will extract the text inside `html_table():` If there is a table in a node, this will extract it.

If there is a table in the node, we can pass the node in this and it will extract the table

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

We will pass in a node object inside this function

It shows the structure of the HTML in a similar way to `prettify` for JSON objects

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

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

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

Example of a function call

`html_attr()` - Extracts out the value of a certain attribute of a node

We will pass in the node and also the `name` of the attribute that we want to from the object

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

Example of a function call

`html_attrs()` - Extracts out all the values of the attributes of a node

We just need to pass in the node and it will extract out all of it

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

Example of a function call

`html_element()` - If we have nested html nodes inside the html node, we can use this

We can pass in the node into function and specify what is the tag that we want to extract

```
html_element(nodes2, "a") # This will extract the a tag from nodes2

Structure of nodes2[1]
[[1]]
<li>
  <a [href, title]>
    {text}
```

`html_children()` - States the element children of the node

We can pass in the node object into this function and it will output the children of the node

3. **Password/Login**

▼ **Usage**

- Sometimes we may need to fill in a form (usually a search box) before the web-page with the data we wish to scrape is generated
- There could be sites that forbid bots from submitting the forms and we need something to mimic a true browser

▼ **Packages**

`rselenium`

`selenium` (Python)

4. **Using REST API**

Package / Tools

`httr` - Package that we can use. Make use of `GET()` and `POST()`

`openauth2`

Usage

- GET/POST requests

GET requests

- When we are requesting things from the website

POST requests

- When we are trying to send something to the website

Functions

`httr`

`GET(base_url, query = list, verbose())` - Allows us to send a request to a website to get information from it

`base_url` - The url in which we will be getting the information from

`query` - This is the query that we want to get from the website. It should be in a list whereby we have the pair of the `query_type = value` . This will modify the URL and supply the query.

`verbose()` - This can be supplied under GET to see what is the status of the request

---

`status_code(GET_result)` - We can get the status code for our GET request if we supply the `verbose()` argument. This can show us if there are any errors in our call

`200` - Ok

`400` - Bad Request

`message_for_status(GET_result)` - Tells us the message for the status code for the GET request

---

`content(GET_result, as)` - Get the content from the GET request provided that the request was successful

- Note that the result will be in a list as compared to a data frame which is formed from `fromJSON`
- `as` - desired type of output: `raw`, `text` or `parsed`. content attempts to automatically figure out which one is most appropriate, based on the content-type.
  - `raw` - Raw object
  - `text` - Character Vector
  - `parsed` - Parsed into a R object

Note

- When we are taking from sites, we can try to see if there are any patterns that we can find to sieve out the URL if needed so that we can reduce redundant things that we need to do