Lecture 14: Data gathering

In [6]:
```
%load_ext rmagic
```

```
The rmagic extension is already loaded. To reload it, use:
  %reload_ext rmagic
```

File and directory manipulations

R offers a bunch of functions to

- **list the files and directories** (see here (http://stat.ethz.ch/R-manual/R-devel/library/base/html/list.files.html)):

```
list.files(path = ".")
list.dirs(path = ".")
```

- **create and modify files** (see here (http://stat.ethz.ch/R-manual/R-devel/library/base/html/files.html) for details):

```
file.create(path1, path2, etc.)
file.remove(path1, path2, etc.)
```

- **create directories** (see here (http://stat.ethz.ch/R-manual/R-devel/library/base/html/files2.html) for details)

```
dir.create(path)
```

- **navigate the file system** (see here (http://stat.ethz.ch/R-manual/R-devel/library/base/html/getwd.html)):

```
getwd()
setwd(dir)
```

Let us create

- a directory `'DATA'` to help us organize our data
- a subdirectory `'raw'` to store the **raw data** collected from the web
- a subdirectory `'cleaned'` to stored our final data after having cleaned the raw data

and then let us set move to the raw directory.

In [2]:
```
%%R
data_dir    = './data'
raw_dir     = paste(data_dir, '/raw', sep='')
cleaned_dir = paste(data_dir, '/cleaned', sep='')
```

```
dir.create(data_dir)
dir.create(raw_dir)
dir.create(cleaned_dir)
```

In [3]:
```
!ls data/
```

```
cleaned raw
```

Downloading files from the web

To download a file from the internet in R and store it on the local file system, one uses the function:

```
download.file(fileUrl, destfile, method="curl")
```

It is also a **good practice to keep track of the date** the data was downloaded, since data from the web may likely be updated. For that, one can

- retrieve the current data with the function `date()`
- include the date as part of the data file name

Example: The city of Baltimore (https://data.baltimorecity.gov/Transportation/Baltimore-Fixed-Speed-Cameras/dz54-2aru) offers an **API** (Application Programming Interface) allowing us to gather data on speed infractions in **different formats**:

- **csv** (Comma Separated Values)
- **xml** (Extensible Markup Language)
- **jason** (JavaScript Object Notation)
- **xls** (Microsoft Excel file format)

To access this different data format, one neeed to use different urls:

In [7]:
```
%%R
csv_url = 'https://data.baltimorecity.gov/api/views/dz54-2aru/rows.csv?accessType=DOWNLOAD'

xml_url = 'https://data.baltimorecity.gov/api/views/dz54-2aru/rows.xml?accessType=DOWNLOAD'

jason_url = 'https://data.baltimorecity.gov/api/views/dz54-2aru/rows.json?accessType=DOWNLOAD'

xls_url = 'https://data.baltimorecity.gov/api/views/dz54-2aru/rows.xls?accessType=DOWNLOAD'
```

Let us prepare prepare the names of the files in which we will store the data:

In [11]:
```
%%R

date = paste(strsplit(date(), split=' ')[[1]], collapse='_')

csv_file     = paste(raw_dir,'/cameras_', date, '.csv', sep='')
xml_file     = paste(raw_dir,'/cameras_', date, '.xml', sep='')
jason_file   = paste(raw_dir,'/cameras_', date, '.jason', sep='')
xls_file     = paste(raw_dir,'/cameras_', date, '.xls', sep='')

print(c(csv_file, xml_file, jason_file, xls_file))
```

```
[1] "./data/raw/cameras_Tue_Nov_19_17:29:05_2013.csv"
[2] "./data/raw/cameras_Tue_Nov_19_17:29:05_2013.xml"
[3] "./data/raw/cameras_Tue_Nov_19_17:29:05_2013.jason"
[4] "./data/raw/cameras_Tue_Nov_19_17:29:05_2013.xls"
```

Let us now download the actual files:

In [12]:
```
%%R

download.file(csv_url,destfile=csv_file, method="curl")
download.file(xml_url,destfile=xml_file, method="curl")
download.file(jason_url,destfile=jason_file, method="curl")
download.file(xls_url,destfile=xls_file, method="curl")

print(list.files(raw_dir))
```

```
[1] "cameras_Tue_Nov_19_17:29:05_2013.csv"
[2] "cameras_Tue_Nov_19_17:29:05_2013.jason"
[3] "cameras_Tue_Nov_19_17:29:05_2013.xls"
[4] "cameras_Tue_Nov_19_17:29:05_2013.xml"
```

Loading data into R

Different data formats are loaded with different functions. The resulting R object storing the data also depends on the format.

Tabular format

Data files in **tabular format** such as **csv** are loaded directly into data frame with the the function:

```
read.table(file, sep, header)
```

In [13]:
```
%%R

df = read.table(csv_file, sep=',', header=TRUE)
print(head(df))
```

```
                          address direction      street crossStreet
```

```
                          address direction      street  crossstreet
1       S CATON AVE & BENSON AVE       N/B   Caton Ave   Benson Ave
2       S CATON AVE & BENSON AVE       S/B   Caton Ave   Benson Ave
3 WILKENS AVE & PINE HEIGHTS AVE       E/B Wilkens Ave Pine Heights
4        THE ALAMEDA & E 33RD ST       S/B The Alameda      33rd St
5        E 33RD ST & THE ALAMEDA       E/B      E 33rd  The Alameda
6        ERDMAN AVE & N MACON ST       E/B      Erdman     Macon St
                intersection                       Location.1
1     Caton Ave & Benson Ave (39.2693779962, -76.6688185297)
2     Caton Ave & Benson Ave (39.2693157898, -76.6689698176)
3 Wilkens Ave & Pine Heights  (39.2720252302, -76.676960806)
4     The Alameda  & 33rd St (39.3285013141, -76.5953545714)
5      E 33rd  & The Alameda (39.3283410623, -76.5953594625)
6         Erdman  & Macon St (39.3068045671, -76.5593167803)
```

XML documents

The tree structure of an xml document

Let us create a simple XML document.

Consider the following data for two students in a given class given in tabular form:

In [14]:
```
%%file course.csv
name, midterm, final, homework, section, major
Bob Durant, 55, 88, 99, 1, STAT
Agnes Thomas, 99, 90, 99, 2, ECON
```

```
Writing course.csv
```

Let's read this data nicely into a data frame:

In [15]:
```
%%R
df = read.csv('course.csv'); print(df)
```

```
          name midterm final homework section major
1   Bob Durant      55    88       99       1  STAT
2 Agnes Thomas      99    90       99       2  ECON
```

The same data can be stored in XML format in the following way:

In []:
```
%%file stat133.xml
<?xml version="1.0" encoding="ISO-8859-1"?>


<stat133 name='Computing with Data' department='STAT'>

    <student sid='1232243'>
        <name>Bob Durant</name>
        <midterm>55</midterm>
```

```
        <final>88</final>
        <homework>99</homework>
        <section>1</section>
        <major>STAT</major>
    </student>

    <student sid='3213453'>
        <name>Agnes Thomas</name>
        <midterm>99</midterm>
        <final>90</final>
        <homework>99</homework>
        <section>2</section>
        <major>ECON</major>
    </student>

</stat133>
```

The first line

```
<?xml version="1.0" encoding="ISO-8859-1"?>
```

provides informations about

- which **version of XML** we are structuring our data with (version 1.0)
- which type of **character encoding** we are using (ISO-8859-1)

The whole course data is surrounded by two **XML tags**

```
<stat133> ... </stat133>
```

- The **first tag** is an **opening tag** indicating the beginning of the stat 133 data:

  ```
  <tag>
  ```

- The **second tag** is a **closing tag** incating the end of the stat 133 data:

  ```
  </tag>
  ```

- Two such tags together with all the data in between tag forms an **xml node**:

  ```
  <tag>...data...</tag>
  ```

The data in between the `stat133` node is itself a **collection of xml nodes**:

```
<student sid='1232243'> ... </student>
<student sid='3213453'> ... </student>
```

- These nodes are the **children nodes** of the **enclosing node** (the `stat133` node here)
- The **enclosing node** is called the **parent node**

Pictorially, one may represent the **children/parent relationship** as an **arrow:**

parent node (stat133 node) $\longrightarrow$ **children node** (student node)

With this interpretation, **an xml document structures data in the form of a tree**, called the **xml tree**.

- The **root** of the tree is the **node whose tag encloses all other nodes**
- The **leafs** of the tree are the **nodes enclosing no other node**

In our class example:

- the **stat133** node is the **root**
- the tags enclosing **actual data** about the students are the **leafs**:

```
<name>Bob Durant</name>
<midterm>55</midterm>
<final>88</final>
<homework>99</homework>
<section>1</section>
<major>STAT</major>
```

So each student node contains five leafs.

The **values** of an **xml node** is the collection of **all the data** contained by **all the leafs stemming from this node**.

Example: Consider the following xml code.

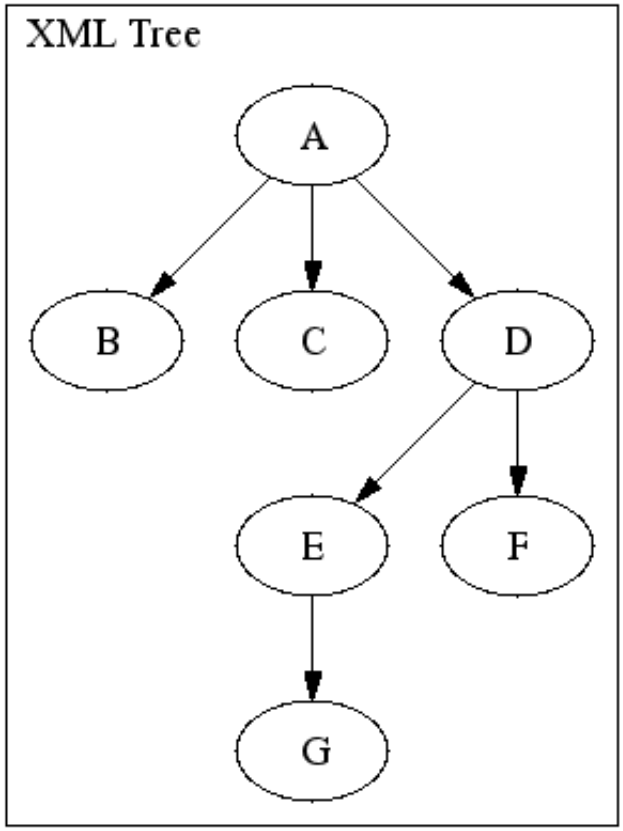
```
<A>
    <B>value1</B>
    <C>value2</C>
    <D>
        <E>
            <G>value3</G>
        </E>
        <F>value4</F>
    </D>
</A>
```

The **value** of

- node A is the collection {value1, value2, value2, value4}
- node B is {value1}
- node D is {value3, value4}
- etc.

Here is a tree representation of the xml code above:

Here is a tree representation of the XML code above:



An **xml node** can also have several **attributes**:

```
<tag attribute1=value attribue2=value etc.>...data...</tag>
```

The **node attributes** are specified in the **opening tag**.

In our class example:

```
<stat133 name='Computing with Data' department='STAT'>

    <student sid='1232243'> ...data... </student>

    <student sid='3213453'> ...data... </student>

</stat133>
```

- the **stat133 node** has **two attribues**: `name` and `department`
- the **student nodes** have only **one attribute**: `sid`

XML node summary

Each **XML node** in a **XML document** may have

- several **attributes** specified in the **opening tag**

- several **children nodes** in between the **opening tag** and the **closing tag**
- one **parent node** enclosing it
- a **value** given by all the data enclosed by the **leafs** steming out of it

The collection of all **XML nodes** with the **parent/children** relation ship form the **XML tree**:

- the **XML node** enclosing all other nodes is the **root** of the **XML document**
- the **XML nodes** enclosing no other nodes are the **leafs** of the **XML document**

Parsing an XML document with R

In R, the **XML library** contains **functions** and **classes** to represent **XML documents**.

Libraries in R corresponds to **modules** in Python.

To **install** the XML library (if not already installed), type in this command:

```
install.packages('XML')
```

Then to use this library type in:

In [16]:
```
%%R
library(XML)
```

```
Attaching package: 'XML'

The following object(s) are masked from 'package:tools':

    toHTML
```

Remark: The keyword `library` in R corresponds to the `import` keyword in Python.

The **XML library** provides two main classes:

A class

```
XMLDocument
```

representing **whole XML documents** including

- The **header of the XML document**, that is, the **data** (also called **meta data**) contained in

  ```
  <?xml version="1.0" encoding="ISO-8859-1"?>
  ```

- The **body of the XML document**, i.e., the **sequence of XML tags representing**

the actual data

A class

```
XMLNode
```

representing the **XML nodes** in the **XML tree** where the information is stored.

The first steps to load XML data into R is to use the functions:

```
xml_doc = xmlParse(file_adress)
root_node = xmlRoot(xml_doc)
```

- `xmlParse` takes as input the XML file address and returns the corresponding `XMLDocument` object
- `xmlRoot` takes an `XMLDocument` and returns the **root node** as a `XMLNode` object.

In [19]:
```
%%R

xml_doc = xmlParse('stat133.xml')
stat133_node = xmlRoot(xml_doc)
```

To **navigate the XML tree** the XML library offers the following functions:

```
xmlAttrs(node)
xmlValue(node)
xmlChildren(node)
xmlParent(node)
```

that take an `XMLNode` object and return the corresponding information.

In [20]:
```
%%R
#returns the attributes as a character vector
stat133_attrs = xmlAttrs(stat133_node)
print(stat133_attrs)
```

```
                name          department
"Computing with Data"               "STAT"
```

In [21]:
```
%%R
#return the children nodes as a list of XMLNode objects
stat133_children = xmlChildren(stat133_node) # return chi

student1_node = stat133_children[[1]]
student2_node = stat133_children[[2]]

#printing an XMLNode yields back the corresponding XML code
print(student1_node)
```

```
<student sid="1232243">
  <name>Bob Durant</name>
```

```
  <midterm>55</midterm>
  <final>88</final>
  <homework>99</homework>
  <section>1</section>
  <major>STAT</major>
</student>
```

In [22]:
```
%%R

parent_node = xmlParent(student1_node)

print(parent_node)
```

```
<stat133 name="Computing with Data" department="STAT">
  <student sid="1232243">
    <name>Bob Durant</name>
    <midterm>55</midterm>
    <final>88</final>
    <homework>99</homework>
    <section>1</section>
    <major>STAT</major>
  </student>
  <student sid="3213453">
    <name>Agnes Thomas</name>
    <midterm>99</midterm>
    <final>90</final>
    <homework>99</homework>
    <section>2</section>
    <major>ECON</major>
  </student>
</stat133>
```

In [23]:
```
%%R

for (student in xmlChildren(stat133_node))
{
    for(node in xmlChildren(student)) print(xmlValue(node))
}
```

```
[1] "Bob Durant"
[1] "55"
[1] "88"
[1] "99"
[1] "1"
[1] "STAT"
[1] "Agnes Thomas"
[1] "99"
[1] "90"
[1] "99"
[1] "2"
[1] "ECON"
```

One can also **retrieve a child node by its tag name**, using the construct:

```
child_node = node[['name']]
```

that returns the **child node** corresponding to `'tag'`.

In [26]:
```
%%R

name_node  = student1_node[['name']]
name_value = xmlValue(name_node)

print(name_value)
```

```
[1] "Bob Durant"
```

The **tag names** of a given **node** can be retrieved using the function

```
names(node)
```

that returns a **character vector** containing the tag names of the children nodes.

Remark: One can do that because an object of any class in R is just an **enhanced list**. Thus we can access some of the methods of the underlying list; here the `name` method that returns the element labels of a list.

In [27]:
```
%%R

tag_names = names(student1_node)
print(tag_names)
```

```
      name    midterm      final   homework    section      major
    "name"  "midterm"    "final" "homework"  "section"    "major"
```

In [28]:
```
%%R

for(student in xmlChildren(stat133_node))
{
    for(tag_name in names(student))
    {
       cat(tag_name, ':', xmlValue(student[[tag_name]]), '\n')
    }
    cat('\n-----------\n')
}
```

```
name : Bob Durant
midterm : 55
final : 88
homework : 99
section : 1
major : STAT

-----------
name : Agnes Thomas
midterm : 99
```

```
final : 90
homework : 99
section : 2
major : ECON

-----------
```

A digression on lists: the double bracket operator

As we saw, the **single bracket operator**

```
list[range]
```

allows us to retrive a **sublist** corresponding to the **range**.

In case, the **range is a single index** then

```
list[i]
```

is a **sublist with one element**, namely the **element at position i**.

CAUTION: A **list with a single element is different that the element itself.**

In [29]:
```
%%R

element1 = c('a', 'b', 'c')   #character vector
element2 = c(1,2)             #numeric vector

my_list  = list(element1, element2)

x = my_list[1]
```

In [30]:
```
%%R

print(class(element1))
print(class(x))
```

```
[1] "character"
[1] "list"
```

In [31]:
```
%%R

print(x[1])
```

```
[[1]]
[1] "a" "b" "c"
```

In [32]:
```
%%R

print(element1[1])
```

```
[1] "a"
```

To **retrieve** the **actual list element** at a given **position i**, one needs to use the **double bracket operator**:

```
list[[i]]
```

This also valid, when one uses **element labels** instead of **position indices**:

```
list[['label']] #retrieves the actual list element
list['label']   #yields back a list containing the list element
```

In [33]:
```
%%R

x = list(names=c('Bob', 'Luc', 'Paul'), grades=c(45, 76,12))
print(x)
```

```
$names
[1] "Bob"  "Luc"  "Paul"

$grades
[1] 45 76 12
```

In [34]:
```
%%R
print(x['names'])
print(x[['names']])
```

```
$names
[1] "Bob"  "Luc"  "Paul"

[1] "Bob"  "Luc"  "Paul"
```

A digression on lists: lapply and sapply

Given a **function** `f` and a **list**

$$x = (x_1, x_2, \ldots, x_n)$$

the **list apply** function:

```
lapply(x, f)
```

will return the **list**

$$(f(x_1), \ldots, f(x_n)).$$

In [35]:
```
%%R
```

```
f = function(x) return(c(mean=mean(x), sd=sd(x)))

var1 = c(1,2,3,4,5,6,8,9)
var2 = c(18,34,2)

x = list(V1=var1, V2=var2)

y = lapply(x, f)

print(y)
```

```
$V1
    mean       sd
4.750000 2.815772

$V2
mean   sd
  18   16
```

Now suppose the function f returns a single number from a numeric vector:

In [36]:
```
%%R

f = function(x) return(mean=mean(x))

y = lapply(x, f)

print(y)
```

```
$V1
[1] 4.75

$V2
[1] 18
```

In this case, it makes sense that the **output** of **lapply** would rather be a **numeric vector** rather than a **list of numeric vectors with one element each**.

For that purpose, one has the **simplify apply function**

```
sapply(x, f, args)
```

that will try to simplify the **output** into a vector of the proper type.

Remark: The last argument `args` can be omitted, but it can also be used to pass additional parameters to the function f, if needed.

In [37]:
```
%%R

y = sapply(x, f)
```

```
print(y)
```

```
   V1    V2
 4.75 18.00
```

From XML document to data frames

Once data has been

- downloaded into the local file system in some given format
- uploaded onto R using the appropriate library

one needs to convert the data representation into **data frames**.

Namely, **data analysis methods** mostly applies to data put **data cubes** or **data frames**.

Data in loaded onto R as **XML documents** are not immediately ready for analysis.

To convert them into **data frame**, one may use complicated `for` loops and branching statements.

A **much better way** is to use **R built-in vectorized capabilities** along with the **apply** function family.

As an example, let's try to transform our **XML object** representation of the class data back into a data frame:

In [38]:
```
%%R

print(stat133_node)
```

```
<stat133 name="Computing with Data" department="STAT">
  <student sid="1232243">
    <name>Bob Durant</name>
    <midterm>55</midterm>
    <final>88</final>
    <homework>99</homework>
    <section>1</section>
    <major>STAT</major>
  </student>
  <student sid="3213453">
    <name>Agnes Thomas</name>
    <midterm>99</midterm>
    <final>90</final>
    <homework>99</homework>
    <section>2</section>
    <major>ECON</major>
  </student>
</stat133>
```

Given

- a list of the **student nodes**
- a **tag name** such as 'final'

one may be able to construct a **vector** representing the the values of corresponding variable.

In [39]:
```
%%R

students = xmlChildren(stat133_node)
```

Let us now write a function that

- takes a **student node** and a **tag name** as input
- returns the **tag value** for the corresponding student

The idea is then to **apply** this function to the **list** of students, we have just constructed:

In [40]:
```
%%R

get_value = function(node, tag) return(xmlValue(node[[tag]]))

name = get_value(students[[1]], 'name')
print(name)
```

```
[1] "Bob Durant"
```

Using this function along with apply, we can now extract the first column or our data frame:

In [42]:
```
%%R

names = sapply(students, get_value, 'name')
print(names)
```

```
       student        student
  "Bob Durant" "Agnes Thomas"
```

We can now package this line of code into a function that takes a list of students and

In [49]:
```
%%R

g = function(tag, student_nodes) return(sapply(student_nodes, get_value, t
ag))

print(g('midterm', students))
```

```
student student
   "55"    "99"
```

In [50]:
```
%%R

tag_names = names(students[[1]])

data = lapply(tag_names, g, students)
```

In [51]:
```
%%R

df = data.frame(data)
print(df)
```

```
          name midterm final homework section major
1   Bob Durant      55    88       99       1  STAT
2 Agnes Thomas      99    90       99       2  ECON
```

In []:

In []:

In []:

In []: