Lecture 10 : Data Acquisition (Part I)

Data Science, DST, UIC

Data is available from a variety of sources but must be retrieved and ultimately processed before it can be used. We can find it in numerous public data sources as

- simple local files
- more complex forms across the Internet.

In this chapter, we will demonstrate how to acquire data from both local files as well as Internets with various data formats. We first starts with reading and writing data in various formats from local files and then introduce web scraping techniques.

Reading and Writing Data in Various Formats

1. Data Formats used in data science applications

When we discuss data formats, we are referring to content format, as opposed to the underlying file format. We cannot examine all available formats due to the vast number of formats available. Instead, we will tackle several of the commonly used formats, providing adequate examples to address the most common data retrieval needs.

Specifically, we will demonstrate how to retrieve data stored in the following formats:

- CSV
- Spreadsheets
- JSON
- XML

The Pandas I/O API is a set of top level *reader* functions accessed like `pandas.read_csv()` that generally return a pandas object. The corresponding *writer* functions are object methods that are accessed like `DataFrame.to_csv()` . Below is a table containing available readers and writers.

Data Format	Reader	Writer
CSV	`read_csv`	`to_csv`
Spreadsheet	`read_excel`	`to_excel`
JSON	`read_json`	`to_json`
XML	-	-

With Python being a popular language for data analysis, it provides various built-in modules for reading and writing data in different formats. Following shows the various data formats and the corresponding module:

Data Format	Module
CSV	`csv`
Spreadsheet	`xlrd` , `xlwt`
JSON	`json`
XML	`ElementTree`

2. CSV

- **Comma Separated Values (CSV)** files, contain tabular data organized in a row-column format.
- The data, stored as plaintext, is stored in rows, also called *records*. Each record contains fields separated by **commas**.

The following is a part of a simple CSV file:

```
city,latd,longd,population_total,area_total,area_land,area_water,area_water_percent
Adelanto,34.57611111,-117.4327778,31765,145.107,145.062,0.046,0.03
AgouraHills,34.15333333,-118.7616667,20330,20.26,20.184,0.076,0.37
Alameda,37.75611111,-122.2744444,75467,59.465,27.482,31.983,53.79
Albany,37.88694444,-122.2977778,18969,14.155,4.632,9.524,67.28
Alhambra,34.08194444,-118.135,83089,19.766,19.763,0.003,0.01
AlisoViejo,33.575,-117.7255556,47823,19.352,19.352,0,0
```

Notice that the first row contains header data to describe the subsequent records. Each value is separated by a comma and corresponds to the header in the same position.

2.1 Read & Write CSV data with `csv` module

Python provides `csv` module to read and write CSV data. The `csv` module implements classes to read and write tabular data in CSV format.

- The `csv` module's `reader` and `writer` objects can read and write sequences.
- You can also read and write data in dictionary form using the `DictReader` and `DictWriter` objects.

The returned object of `csv.reader` and `csv.DictReader` are iterators. Each iteration returns a row (sequence or dictionary) of the CSV file.

`with` statement in Python is used in exception handling to make the code cleaner and much more readable.

- `with` statement helps avoiding bugs and leaks by ensuring that a resource is properly released when the code using the resource is completely executed.
- The `with` statement is popularly used with file streams, as shown below and with Locks, sockets, subprocesses and telnets etc.

In [1]:
```
import csv
# open() :Open file and return a stream.
# csv.reader(): The returned object is an iterator.  Each iteration returns a row of the CSV fil
with open('sample.csv','r') as csvfile:
    freader = csv.reader(csvfile)
    for row in freader:
#         print (row)
        print(row[1:3])
```

```
['Name', 'Age']
['Anna', '12']
['Bob', '13']
['Charles', '8']
['Damon', '9']
```

In [2]:
```
with open('sample.csv','r') as csvfile:
    dreader = csv.DictReader(csvfile)
    for row in dreader:
        print(row)
        print(row['Name'],row['Age'])
```

```
OrderedDict([('', '0'), ('Name', 'Anna'), ('Age', '12')])
Anna 12
OrderedDict([('', '1'), ('Name', 'Bob'), ('Age', '13')])
Bob 13
OrderedDict([('', '2'), ('Name', 'Charles'), ('Age', '8')])
Charles 8
OrderedDict([('', '3'), ('Name', 'Damon'), ('Age', '9')])
Damon 9
```

The csv module can also read and write tabular data in txt file as long as the format is CSV.

In [3]:
```
with open('sample.txt','r') as csvfile:
    dreader = csv.DictReader(csvfile)
    for row in dreader:
        print(row)
        print(row['Name'],row['Age'])
```

```
OrderedDict([('', '0'), ('Name', 'Anna'), ('Age', '12')])
Anna 12
OrderedDict([('', '1'), ('Name', 'Bob'), ('Age', '13')])
Bob 13
OrderedDict([('', '2'), ('Name', 'Charles'), ('Age', '8')])
Charles 8
OrderedDict([('', '3'), ('Name', 'Damon'), ('Age', '9')])
Damon 9
```

Writing to CSV file can be performed using `csv.writer`:

In [6]:
```
header = ['Name','Age']
rows = [['Anna',13],['Bob',12],['Charles',8],['Damon','9']]
with open('writeCSV_sample.csv', 'w') as csvfile:
    writer = csv.writer(csvfile)
    writer.writerow(header)
    writer.writerows(rows)
```

In [7]:
```
with open('writeCSV_sample.csv','r')as csvfile:
    freader = csv.reader(csvfile)
    for row in freader:
        print(row)
```

```
['Name', 'Age']
[]
['Anna', '13']
[]
['Bob', '12']
[]
['Charles', '8']
[]
['Damon', '9']
[]
```

In [6]:
```
dict_data = [{'Name':'Anna','Age':'13'},{'Name':'Bob','Age':'12'}]
with open('writeDictCSV_sample.csv', 'w', newline='') as csvfile:
    fieldnames = ['Name', 'Age']
    writer = csv.DictWriter(csvfile, fieldnames=fieldnames)
    writer.writeheader()
    writer.writerows(dict_data)
    writer.writerow({'Name': 'Charles', 'Age': '8'})
    writer.writerow({'Name': 'Damon', 'Age': '9'})
```

In [7]:
```
csvfile = open('writeDictCSV_sample.csv','r')
freader = csv.DictReader(csvfile)
for row in freader:
    print(row)
csvfile.close()
```

```
OrderedDict([('Name', 'Anna'), ('Age', '13')])
OrderedDict([('Name', 'Bob'), ('Age', '12')])
OrderedDict([('Name', 'Charles'), ('Age', '8')])
OrderedDict([('Name', 'Damon'), ('Age', '9')])
```

- `writerow(row)` method writes the *row* parameter to the writer's file object.
- `writerows(rows)` writes all elements in *rows* to the writer's file object.

2.2 Read & Write CSV data with Pandas

In [8]:
```
import pandas as pd
cal_data = pd.read_csv("california_cities.csv")
cal_data.head()
```

Out[8]:

	city	latd	longd	population_total	area_total	area_land	area_water	area_water_perce
0	Adelanto	34.576111	-117.432778	31765	145.107	145.062	0.046	0.
1	AgouraHills	34.153333	-118.761667	20330	20.260	20.184	0.076	0.
2	Alameda	37.756111	-122.274444	75467	59.465	27.482	31.983	53.
3	Albany	37.886944	-122.297778	18969	14.155	4.632	9.524	67.
4	Alhambra	34.081944	-118.135000	83089	19.766	19.763	0.003	0.

The `read_csv` function takes the path of the `.csv` file to input the data. To write a data to the `.csv` file, we can use `to_csv` function of `pandas`:

In [9]:
```
w_data = {'Name':['Anna','Bob','Charles','Damon'],'Age':[12,13,8,9] }
df_data = pd.DataFrame(w_data)
df_data.to_csv('sample_1.csv')
```

The `DataFrame` is written to a `.csv` file by using the `to_csv` method. The path and the filename where the file needs to be created should be mentioned.

In [10]:
```
r_data = pd.read_csv('sample_1.csv')
r_data
```

Out[10]:

	Unnamed: 0	Name	Age
0	0	Anna	12
1	1	Bob	13
2	2	Charles	8
3	3	Damon	9

Here, the `to_csv` function will write row names (index) by default. There are two ways to handle the situation where we do not want the index to be stored in csv file

- You can use `index = False` while saving dataframe to csv file.

In [11]:
```
df_data.to_csv('sample_WT_Index.csv', index = False)
r_data = pd.read_csv('sample_WT_Index.csv')
r_data
```

Out[11]:

	Name	Age
0	Anna	12
1	Bob	13
2	Charles	8
3	Damon	9

- Or you can save your dataframe as it is with index. While reading, just drop the column containing the index.

In [12]:
```
r_data = pd.read_csv('sample_1.csv').drop(['Unnamed: 0'],axis=1)
r_data
```

Out[12]:

	Name	Age
0	Anna	12
1	Bob	13
2	Charles	8
3	Damon	9

- You can also read the data with the index by using `index_col='Unnamed: 0'` :

In [13]:
```
r_data = pd.read_csv('sample_1.csv', index_col='Unnamed: 0')
r_data
```

Out[13]:

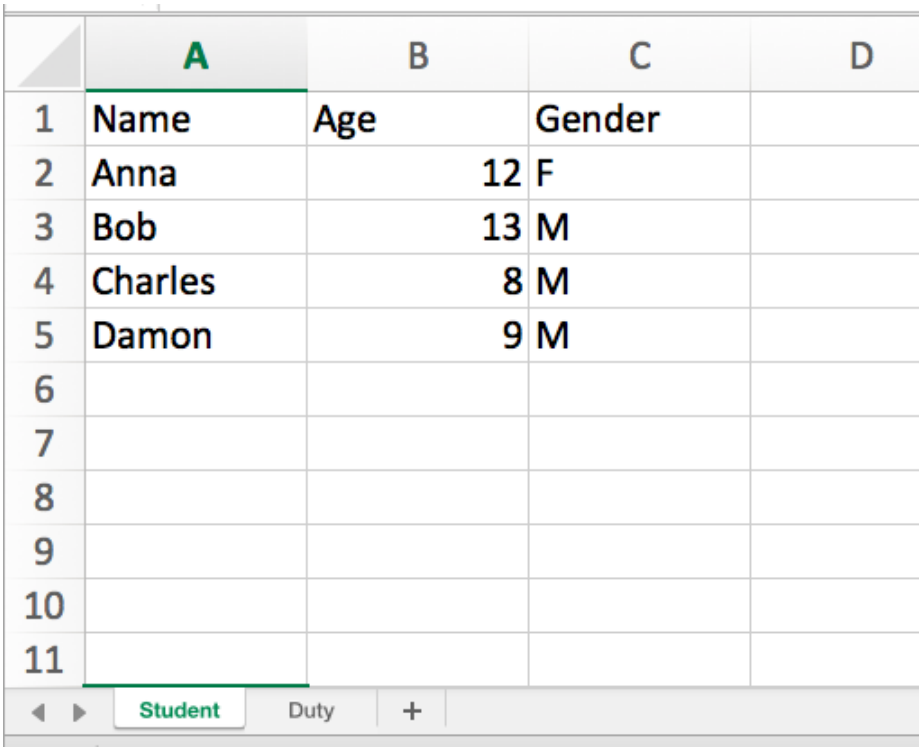
	Name	Age
0	Anna	12
1	Bob	13
2	Charles	8
3	Damon	9

As the `read_csv` function returns a `DataFrame` object. You can process and manage the data as introduced in Pandas chapter.

3. Spreadsheets

Spreadsheets are a form of tabular data where information is stored in rows and columns, much like a two-dimensional array. They typically contain numeric and textual information and use formulas to summarize and analyze their contents.

Spreadsheets are an important data source because they have been used for the past several decades to store information in many industries and applications. Their tabular nature makes them easy to process and analyze. It is important to know how to extract data from this ubiquitous data source so that we can take advantage of the wealth of information that is stored in them.



	A	B	C	D
1	Name	Age	Gender	
2	Anna	12	F	
3	Bob	13	M	
4	Charles	8	M	
5	Damon	9	M	
6				
7				
8				
9				
10				
11				

Student Duty +

3.1 Read & Write Spreadsheet Data with `xlrd` and `xlwt` modules

Similarly, python also provides a `xlrd` module to extract data from spreadsheet and `xlwt` module to write to spreadsheet files.

In [14]:
```
import xlrd
wb = xlrd.open_workbook('students.xlsx')
sheet = wb.sheet_by_index(0)
# sheet = wb.sheet_by_name('Student')
# get cell value
print(sheet.cell_value(0,0))

# get row value
print(sheet.row_values(0))

# get value from each row
for i in range(sheet.nrows):
    print(sheet.row_values(i))
```

```
Name
['Name', 'Age', 'Gender']
['Name', 'Age', 'Gender']
['Anna', 12.0, 'F']
['Bob', 13.0, 'M']
['Charles', 8.0, 'M']
['Damon', 9.0, 'M']
```

In [15]:
```
import xlwt
# Workbook is created
wb = xlwt.Workbook()

# add_sheet is used to create sheet.
sheet1 = wb.add_sheet('student_info')

sheet1.write(0, 0, 'Name')
sheet1.write(1, 0, 'Anna')
sheet1.write(2, 0, 'Bob')
sheet1.write(3, 0, 'Charles')
sheet1.write(4, 0, 'Damon')
sheet1.write(0, 1, 'Age')
sheet1.write(1, 1, '13')
sheet1.write(2, 1, '12')
sheet1.write(3, 1, '8')
sheet1.write(4, 1, '9')

wb.save('writeExcel_sample.xlsx')
```

3.2 Read & Write Spreadsheet with Pandas

Excel files can be read using the Python module Pandas. The method `read_excel()` reads the data into a Pandas DataFrame, where the first parameter is the filename and the second parameter is the sheet name.

In [16]:
```
# import xlrd
ex_data = pd.read_excel('writeExcel_sample.xlsx')
ex_data
```

Out[16]:

	Name	Age
0	Anna	13
1	Bob	12
2	Charles	8
3	Damon	9

In [17]:
```
duty_data = pd.read_excel('students.xlsx', sheet_name = 'Duty')
duty_data
```

Out[17]:

	Student	Day
0	Bob	Monday
1	Anna	Tuesday
2	Charles	Wednesday
3	Damon	Thursday
4	Bob	Friday

To write data into spreadsheets format, you can use `to_excel` function. `to_excel` serializes lists and dicts to strings before writing. Once a workbook has been saved, it is not possible write further data without rewriting the whole workbook.

In [18]:
```
df1 = pd.DataFrame([['a','b'],['c','d']], index=['row1','row2'], columns=['col1','col2'])
df1.to_excel("sampleExcel.xlsx", sheet_name='testsheet')
```

In [19]:
```
df2 = pd.read_excel("sampleExcel.xlsx")
df2
```

Out[19]:

	col1	col2
row1	a	b
row2	c	d

3. JSON

- JSON (*JavaScript Object Notation*) is language-neutral data interchange format. It was created and popularized by Douglas Crockford. In its short history, JSON has become a defacto standard for data transfer across the web.
- JSON is commonly used by web applications to transfer data between client and server. If you are using a web service then there are good chances that data will be returned to you in JSON format, by default.

JSON syntax is derived from JavaScript object notation syntax:

- Data is in name/value pairs
- Data is separated by commas
- Curly braces hold objects
- Square brackets hold arrays

For example: `[{"Name":"Anna", "Age":"13"},{"Name":"Bob","Gender": "M"}]` is json format holding two objects, each object has two name/value pairs. Note that in JSON, **string values must be written with double quotes**.

3.1 Read & Write JSON Data with `json` Module

Python comes with a built-in package called `json` for encoding and decoding JSON data.

- **Serialization**: The process of converting an object into a special format which is suitable for transmitting over the network or storing in file or database.
- **Deserialization**:It is the reverse of serialization. It converts the special format returned by the serialization back into a usable object.

In the case of JSON, when we serializing objects, we essentially convert a Python object into a JSON string and deserialization builds up the Python object from its JSON string representation. The `json` module mainly provides the following functions for serialization and deserialization:

- `dump(obj, fileobj)`
- `dumps(obj)`
- `load(fileobj)`
- `loads(str)`

Serializing with `dump()`

The `dump(obj,fileobj)` function is used to serialize data. It takes a Python object, serializes it and writes the output (which is a JSON string) to a file like object.

In [20]:
```
import json
person = {"first_name":"John", "isAlive":True, "age":27, "address":{
    "city":"New York","state":"NY"}, "hasMortage":None}
with open('dumpJson_person.json','w') as f:
    json.dump(person,f)

# f.read(size) reads some quantity of data and returns it as a string.
# When size is omitted or negative, the entire contents of the file will be read and returned;
open('dumpJson_person.json', 'r').read()
```

Out[20]:
```
'{"first_name": "John", "isAlive": true, "age": 27, "address": {"city": "New York", "state":
"NY"}, "hasMortage": null}'
```

Notice that while serializing the object, the Python's type `None` is converted to JSON's `null` type. The following table lists the conversion happens between types when we seralize the data. When we deserialize object, JSON type is converted back to its equivalent Python type.

Python Type	JSON Type
dict	object
list,tuple	array
int	number
float	number
str	string
True	true
False	false
None	null

Following is another example which serializes a list of two persons:

In [21]:
```
persons = [{"first_name":("John","Smith"), "isAlive":True, "age":27, "address":{
    "city":"New York","state":"NY"}, "hasMortage":None},
          {"first_name":"Bob", "isAlive":True, "age":32, "address":{
    "city":"Ocean Springs","state":"Mississippi"}, "hasMortage":True}
          ]
with open('dumpJson_persons.json','w') as f:
    json.dump(persons,f)
open('dumpJson_persons.json', 'r').read()
```

Out[21]:
```
'[{"first_name": ["John", "Smith"], "isAlive": true, "age": 27, "address": {"city": "New Yor
k", "state": "NY"}, "hasMortage": null}, {"first_name": "Bob", "isAlive": true, "age": 32, "a
ddress": {"city": "Ocean Springs", "state": "Mississippi"}, "hasMortage": true}]'
```

Deserializing with load()

To deserizalize the Python objects back, we use `load()` function. Its syntax is as follows: `load(fp)->python object .`

In [22]:
```
with open('dumpJson_person.json','r') as f:
    person = json.load(f)
type(person)
```

Out[22]:
```
dict
```

In [23]:
```
print(person)
print(person["first_name"])
print(person["age"])
```

```
{'first_name': 'John', 'isAlive': True, 'age': 27, 'address': {'city': 'New York', 'state':
'NY'}, 'hasMortage': None}
John
27
```

In [24]:
```
with open('dumpJson_persons.json', 'r') as f:
    persons = json.load(f)
type(persons)
```

Out[24]:
```
list
```

In [25]:
```
print(persons)
persons[0]["first_name"]
```

```
[{'first_name': ['John', 'Smith'], 'isAlive': True, 'age': 27, 'address': {'city': 'New Yor
k', 'state': 'NY'}, 'hasMortage': None}, {'first_name': 'Bob', 'isAlive': True, 'age': 32, 'a
ddress': {'city': 'Ocean Springs', 'state': 'Mississippi'}, 'hasMortage': True}]
```

Out[25]:
```
['John', 'Smith']
```

Serializing and Deserializing with `dumps()` and `loads()`

- The `dumps()` function works exactly like `dump()` but instead of sending the output to a file-like object, it returns the output as a string.
- Similarly, `loads()` function is as same as `load()` but instead of deserializing the JSON string from a file, it deserializes from a string.

In [26]:
```
person
```

Out[26]:
```
{'first_name': 'John',
 'isAlive': True,
 'age': 27,
 'address': {'city': 'New York', 'state': 'NY'},
 'hasMortage': None}
```

In [27]:
```
data = json.dumps(person)
data
```

Out[27]:
```
'{"first_name": "John", "isAlive": true, "age": 27, "address": {"city": "New York", "state":
"NY"}, "hasMortage": null}'
```

In [28]:
```
person_n = json.loads(data)
print(type(person_n))
person
```

```
<class 'dict'>
```

Out[28]:
```
{'first_name': 'John',
 'isAlive': True,
 'age': 27,
 'address': {'city': 'New York', 'state': 'NY'},
 'hasMortage': None}
```

3.2 Read & Write JSON Data with Pandas

Next, we show how to read and write json with pandas library:

In [29]:
```
json_text = '[{"Name":"Anna", "Age":13},{"Name":"Bob","Gender": "M"}]'
df1 = pd.read_json(json_text)
df1
```

Out[29]:

	Age	Gender	Name
0	13.0	NaN	Anna
1	NaN	M	Bob

In [30]:
```
json_text = '{"Name":{"FirstName":"Anna","LastName":"M"},"Age":13}'
df2 = pd.read_json(json_text, typ="series")
# df2 = pd.read_json(json_text, typ="frame")
df2
```

Out[30]:
```
Name    {'FirstName': 'Anna', 'LastName': 'M'}
Age                                        13
dtype: object
```

In [31]:
```
# write to json file
df1.to_json("writeJson_sample.json")
pd.read_json("writeJson_sample.json")
```

Out[31]:

	Age	Gender	Name
0	13.0	None	Anna
1	NaN	M	Bob

4. XML

XML (Extensible Markup Language), is a markup-language that is commonly used to structure, store, and transfer data between systems.

Basically, the XML format data are organized in **elements**, which define the type of information exposed, and each element contains the actual data in the form of content or **attributes**. Each piece of information is delimited by a specific **tag**:

```
<data>
    <student name="Anna">
        <email>anna@mail.com</email>
        <grade>A</grade>
        <age>13</age>
    </student>
    <student name="Bob">
        <email>bob@mail.com</email>
        <grade>B</grade>
        <age>12</age>
    </student>
    <student name="Charles">
        <email>charles@mail.com</email>
        <grade>C</grade>
        <age>8</age>
    </student>
    <student name="Damon">
        <email>damon@mail.com</email>
        <grade>A</grade>
        <age>9</age>
    </student>
</data>
```

In this example, each student is represented by a `<student>...</student>` element, which has a name attribute containing the name of a specific student. Each of these elements has then `sub-elements` defined by the `<email>`, `<grade>` and `<age>` tags; between these tags the actual data content referring to the given student is present. Let's say this data is saved in an XML file called "students.xml".

4.1 Read & Write XML Data with `ElementTree` module

Given the structure of XML file, we can represent them as a tree, and this is the approach used by the `xml.etree.ElementTree` Python module. The parsing of our "students.xml" file starts at the root of the tree, namely the `<data>` element, which contains the entire data structure.

In [32]:
```
import xml.etree.ElementTree as et

xtree = et.parse("students.xml")
xroot = xtree.getroot()
xroot
```

Out[32]:
```
<Element 'data' at 0x1bcd550ff60>
```

Element has some useful methods that help iterate recursively over all the sub-tree below it (its children, their children, and so on). For example, Element.iter():

In [33]:
```
for i in xroot.iter('email'):
    print(i.tag, i.text)
```

```
email anna@mail.com
email bob@mail.com
email charles@mail.com
email damon@mail.com
```

- `Element.findall()` finds only elements with a tag which are direct children of the current element;
- `Element.find()` finds the first child with a particular tag;
- `Element.text` accesses the element's text content;
- `Element.get()` accesses the element's attributes.

In [34]:
```
for student in xroot.findall('student'):
    name = student.get('name')
    email = student.find('email').text
    grade = student.find('grade').text
    age = student.find('age').text
    print(name,email,grade,age)
```

```
Anna anna@mail.com A 13
Bob bob@mail.com B 12
Charles charles@mail.com C 8
Damon damon@mail.com A 9
```

We can think of XML data structure as a pandas DataFrame in which each student represents an observation, with its name attribute being the main identifier and the sub-elements being other features of the observation. Now we can iterate through each node of the tree, which means we will get each student element and grab its name attribute and all of its sub-elements to build our dataframe.

In [35]:
```
name_l = []
email_l = []
grade_l = []
age_l = []
for node in xroot:
    s_name = node.get('name')
    s_email = node.find('email').text
    s_grade = node.find('grade').text
    s_age = node.find('age').text

    name_l.append(s_name)
    email_l.append(s_email)
    grade_l.append(s_grade)
    age_l.append(s_age)

s_df = pd.DataFrame({'name':name_l, 'email':email_l, 'grade':grade_l, 'age':age_l})
s_df
```

Out[35]:

	name	email	grade	age
0	Anna	anna@mail.com	A	13
1	Bob	bob@mail.com	B	12
2	Charles	charles@mail.com	C	8
3	Damon	damon@mail.com	A	9

The `ElementTree` module provides limited support for **XPath** expressions for locating elements in a tree. XPath (XML Path Language) is a path language for defining parts of an XML document.

In [36]:
```
import xml.etree.ElementTree as et

xtree = et.parse("students.xml")
xroot = xtree.getroot()
# Top-level elements
print(xroot.findall("."))

# All 'grade' grand-children of 'student' children of the top-level elements
gnodes = (xroot.findall("./student/grade"))
for g in gnodes:
    print(g.text)

# All 'email' nodes
enodes = (xroot.findall(".//email"))
for e in enodes:
    print (e.text)
# All 'student' nodes that are the second child of their parent
snodes = (xroot.findall(".//student[2]"))
print(snodes[0].get('name'))

# Email nodes with student name='Bob'
emailnodes = xroot.findall("*[@name='Bob']/email")
print(emailnodes[0].text)

# Student nodes with grade A
snodes_a = xroot.findall(".//student/[grade='A']")
print('Students with grade A:')
for ss in snodes_a:
    print(ss.attrib.get('name'))
```

```
[<Element 'data' at 0x1bcd555dc18>]
A
B
C
A
anna@mail.com
bob@mail.com
charles@mail.com
damon@mail.com
Bob
bob@mail.com
Students with grade A:
Anna
Damon
```

Supported XPath syntax:

Syntax	Meaning
`tagname`	Selects all child elements with the given tag.For example, `student` selects all child elements named student and `student/grade` selects all grandchildren named grade in all children named student.
`*`	Selects all child elements. For example, `*/email` selects all grandchildren named email.
`.`	Selects the current node. This is mostly useful at the beginning of the path, to indicate that it's a relative path.
`//`	Selects all subelements, on all levels beneath the current element. For example, `.//email` selects all email elements in the entire tree.
`..`	Selects the parent element.
`[@attrib]`	Selects all elements that have the given attribute.
`[@attrib='value']`	Selects all elements for which the given attribute has the given value.
`[tag]`	Selects all elements that have a child named tag. Only immediate children are supported.
`[tag='text']`	Selects all elements that have a child named tag whose complete text content, including descendants, equals the given text.

Syntax	Meaning
[position]	Selects all elements that are located at the given position.The position can be either an integer (1 is the first position),

To write data into XML files, follow the given steps:

- Create an element using `Element(tag)` function, which will act as the root element.
- Create sub-elements by using `SubElement(parent,tag, attrib={})` , where the
  - `parent` is the parent node,
  - `tag` is the tag name,
  - `attrib` is a dictionary containing the element attributes,
  - Write the created data to file.

In [37]:
```
from xml.dom import minidom

data = et.Element('data')
for i in range(0, len(s_df)):
    student = et.SubElement(data,'student',{'name':s_df.iloc[i]['name']})
    email = et.SubElement(student,'email').text=s_df.iloc[i]['email']
    grade = et.SubElement(student,'grade').text=s_df.iloc[i]['grade']
    age = et.SubElement(student,'age').text=s_df.iloc[i]['age']

# This will write to file without pretty printing
tree = et.ElementTree(data)
tree.write('writeXML.xml', xml_declaration=True)
with open('writeXML.xml','r') as re:
    print(re.read())

# Pretty xml formats
xmlstring =  minidom.parseString(et.tostring(data, encoding='UTF-8')).toprettyxml(indent="    ")
with open('writePrettyXML.xml','w') as f:
    f.write(xmlstring)

print(open('writePrettyXML.xml','r').read())
```

```
<?xml version='1.0' encoding='us-ascii'?>
<data><student name="Anna"><email>anna@mail.com</email><grade>A</grade><age>13</age></stud
ent><student name="Bob"><email>bob@mail.com</email><grade>B</grade><age>12</age></student>
<student name="Charles"><email>charles@mail.com</email><grade>C</grade><age>8</age></stude
nt><student name="Damon"><email>damon@mail.com</email><grade>A</grade><age>9</age></studen
t></data>
<?xml version="1.0" ?>
<data>
    <student name="Anna">
        <email>anna@mail.com</email>
        <grade>A</grade>
        <age>13</age>
    </student>
    <student name="Bob">
        <email>bob@mail.com</email>
        <grade>B</grade>
        <age>12</age>
    </student>
    <student name="Charles">
```