

# COMP41680

## Data Formats and Collection

**Derek Greene**

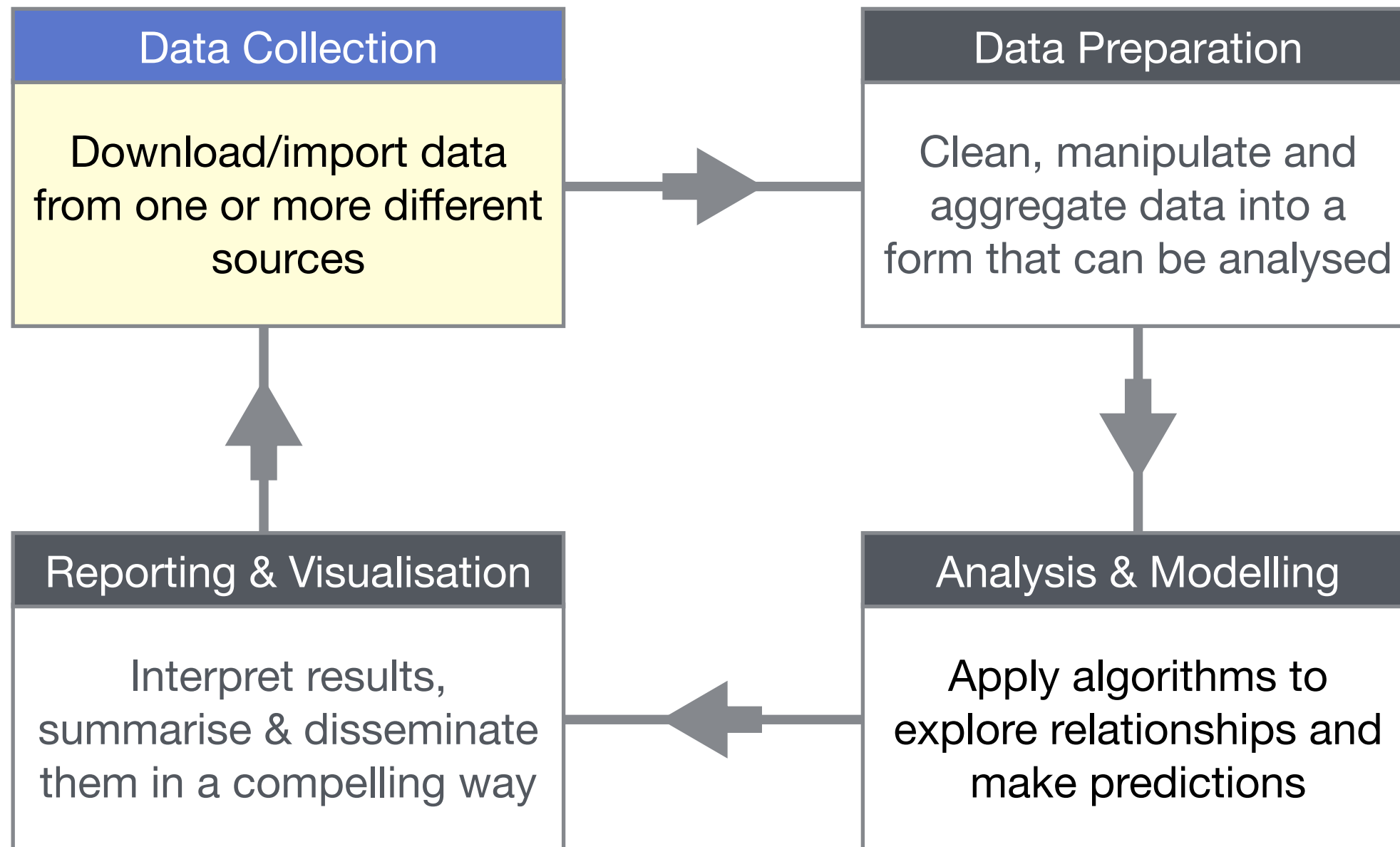
**UCD School of Computer Science**  
**Spring 2023**



# Reminder: Data Science Pipeline

---

- Recall the stages of the basic data science pipeline
- We start the pipeline by acquiring the data that we need...

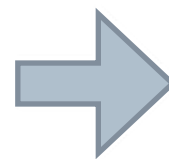


# Data Formats: CSV

- Data takes different forms in different domains (e.g. finance, healthcare, manufacturing) and is stored in a variety of formats.
- **CSV ("Comma Separated Values")**: a file format often used to exchange tabular data between different applications (e.g. Excel).
- Essentially a plain text file where values are split by a comma. Alternatively can use a different separator (e.g. tab, semi-colon).
- Generally includes a header line with column names. The first value on each line is also sometimes a unique identifier for that row.

```
code,name,population,life_expect
BEL,Belgium,11338476,80.99
ESP,Spain,46484533,82.83
FIN,Finland,5495303,81.78
FRA,France,66892205,82.27
IRL,Ireland,4749777,81.61
ITA,Italy,60627498,82.54
LVA,Latvia,1959537,74.53
NOR,Norway,5236151,82.51
POL,Poland,37970087,77.45
SWE,Sweden,9923085,82.20
```

countries10.csv



code	name	population	life_expect
BEL	Belgium	11338476	80.99
ESP	Spain	46484533	82.83
FIN	Finland	5495303	81.78
FRA	France	66892205	82.27
IRL	Ireland	4749777	81.61
ITA	Italy	60627498	82.54
LVA	Latvia	1959537	74.53
NOR	Norway	5236151	82.51
POL	Poland	37970087	77.45
SWE	Sweden	9923085	82.20

Dataset of 10 rows (+ header) and 4 columns



# Data Formats: CSV

- The built-in module **csv** provides an easy way to read and write data in CSV format in Python.

```
import csv
fin = open("countries10.csv", "r")
reader = csv.DictReader(fin)
for row in reader:
    print(row)
fin.close()
```

Read each line (row) in the CSV file  
into a separate dictionary

```
{'code': 'BEL', 'name': 'Belgium', 'population': '11338476', 'life_expect': '80.99'}
{'code': 'ESP', 'name': 'Spain', 'population': '46484533', 'life_expect': '82.83'}
{'code': 'FIN', 'name': 'Finland', 'population': '5495303', 'life_expect': '81.78'}
{'code': 'FRA', 'name': 'France', 'population': '66892205', 'life_expect': '82.27'}
{'code': 'IRL', 'name': 'Ireland', 'population': '4749777', 'life_expect': '81.61'}
{'code': 'ITA', 'name': 'Italy', 'population': '60627498', 'life_expect': '82.54'}
{'code': 'LVA', 'name': 'Latvia', 'population': '1959537', 'life_expect': '74.53'}
{'code': 'NOR', 'name': 'Norway', 'population': '5236151', 'life_expect': '82.51'}
{'code': 'POL', 'name': 'Poland', 'population': '37970087', 'life_expect': '77.45'}
{'code': 'SWE', 'name': 'Sweden', 'population': '9923085', 'life_expect': '82.20'}
```

```
fout = open("output.csv", "w")
fields = ["code", "name", "population", "life_expect"]
writer = csv.DictWriter(fout, fieldnames=fields)
writer.writeheader()
for row in data:
    writer.writerow(row)
fout.close()
```

Write each row in the data out  
to a separate line in a new CSV  
file. Note we need to specify  
the names of our fields.

# Data Formats: JSON

---

- **JSON (JavaScript Object Notation)**: a lightweight format which is becoming increasingly popular for online data exchange. Based originally on the JavaScript language and (relatively) easy for humans to read and write - <http://www.json.org>
- JSON files are built from two different structures:
  1. **Object**: Collection of name/value pairs - like a Python dictionary.  
Begins with { and ends with }  
Each name is followed by : (colon) and the name/value pairs are separated by commas
  2. **Array**: A list of values separated by commas - like a Python list.  
Begins with [ and ends with ]
- Values can be of the following types:
  - String in double quotes; a number; true/false; null; object; array.

# Data Formats: JSON

---

- **Object:** A collection of name/value pairs, separated by commas. It acts like a Python dictionary.

```
{ "firstname": "Alison", "age": 30, "registered": true }
```

```
{  
  "name": "School of Computer Science",  
  "link": "http://www.cs.ucd.ie"  
}
```

- **Array:** A list of values separated by commas - like a Python list. The values can have the same or different types.

```
[ "Ford", "BMW", "Fiat", "Mercedes" ]
```

```
[ 1123, 353, 412, 99.0, "Dublin", false, true ]
```

- We can mix Objects and Arrays:

```
{ "codes" : [5,11,31,41] }
```

An array inside an object.

```
[ { "name": "Alison" }, { "name": "John" } ]
```

Two objects in an array.

# Data Formats: JSON

- The outmost value in a JSON file can be an array or an object.

Top level value is an array, which itself contains two objects.

```
[
  {
    "id" : "978-1933988177",
    "category" : ["book", "paperback"],
    "name" : "Lucene in Action",
    "author" : "Michael McCandless",
    "genre" : "technology",
    "price" : 30.50,
    "pages" : 475
  },
  {
    "id" : "978-1857995879",
    "category" : ["book", "paperback"],
    "name" : "Sophie's World",
    "author" : "Jostein Gaarder",
    "sequence_i" : 1,
    "genre" : "fiction",
    "price" : 3.07,
    "pages" : 64
  }
]
```

Top level value is an object, which contains an array of objects.

```
{
  "students": [{
    "name": "John",
    "age": "23",
    "city": "Dublin"
  }, {
    "name": "Sarah",
    "age": "28",
    "city": "Cork"
  }, {
    "name": "Peter",
    "age": "32",
    "city": "Galway"
  }, {
    "name": "Alice",
    "age": "28",
    "city": "London"
  }]
}
```

# Data Formats: JSON

- Note that whitespace does not matter in JSON, it only makes the data more readable for humans.

```
[
  {
    "id" : "978-1933988177",
    "category" : ["book","paperback"],
    "name" : "Lucene in Action",
    "author" : "Michael McCandless",
    "genre" : "technology",
    "price" : 30.50,
    "pages" : 475
  },
  {
    "id" : "978-1857995879",
    "category" : ["book","paperback"],
    "name" : "Sophie's World",
    "author" : "Jostein Gaarder",
    "sequence_i" : 1,
    "genre" : "fiction",
    "price" : 3.07,
    "pages" : 64
  }
]
```

```
[ { "id" : "978-1933988177", "category" : ["book","paperback"],
  "name" : "Lucene in Action", "author" : "Michael McCandless",
  "genre" : "technology", "price" : 30.50, "pages" : 475 }, { "id"
: "978-1857995879", "category" : ["book","paperback"], "name" :
"Sophie's World", "author" : "Jostein Gaarder", "sequence_i" :
1, "genre" : "fiction", "price" : 3.07, "pages" : 64 } ]
```

```
{
  "students": [{
    "name": "John",
    "age": "23",
    "city": "Dublin"
  }, {
    "name": "Sarah",
    "age": "28",
    "city": "Cork"
  }, {
    "name": "Peter",
    "age": "32",
    "city": "Galway"
  }, {
    "name": "Alice",
    "age": "28",
    "city": "London"
  }]
}
```

```
{"students": [{ "name": "John", "age":
"23", "city": "Dublin"}, { "name":
"Sarah", "age": "28", "city": "Cork"}, { "name":
"Peter", "age": "32", "city": "Galway"},
{ "name": "Alice", "age": "28", "city":
"London" } ]}
```



# Using JSON in Python

- JSON is language-agnostic. Many different parsers available.
- The built-in module `json` provides an easy way to encode and decode data in JSON in Python. Two main functions:

`json.dumps()`: Turn a Python data structure into a JSON string

`json.loads()`: Load a Python data structure from a JSON string

```
import json
names = ["Steve", "Maria", "Jenny"]
s = json.dumps(names)
print(s)
```

```
["Steve", "Maria", "Jenny"]
```

Convert simple list into JSON

```
import json
d = {"Steve":25, "Linda":40, "John":33}
s = json.dumps(d)
print(s)
```

```
{"Linda": 40, "Steve": 25, "John": 33}
```

Convert simple dictionary into JSON

```
products = []
p1={"id":43,"name":"iPhone XR","brand":"Apple"}
products.append(p1)
p2={"id":87,"name":"iPhone 11","brand":"Apple"}
products.append(p2)
s = json.dumps(products)
print(s)
```

```
[{"brand": "Apple", "id": 43, "name": "iPhone XR"}, {"brand": "Apple", "id": 87, "name": "iPhone 11"}]
```

2 dictionaries nested in a list  
converted to JSON

# Using JSON in Python

- To translate a string containing JSON data into a Python value, pass it to the `json.loads()` function.
- Python automatically converts JSON values into variables of the appropriate type - i.e. arrays become lists, objects become dictionaries, numbers become integers or floats.

File: books.json

```
[
  {
    "id" : "978-1933988177",
    "category" : ["book","paperback"],
    "name" : "Lucene in Action",
    "author" : "Michael McCandless",
    "genre" : "technology",
    "price" : 30.50,
    "pages" : 475
  },
  {
    "id" : "978-1857995879",
    "category" : ["book","paperback"],
    "name" : "Sophie's World",
    "author" : "Jostein Gaarder",
    "genre" : "fiction",
    "price" : 3.07,
    "pages" : 518
  }
]
```

```
import json
fin = open("books.json","r")
s = fin.read()
fin.close()
data = json.loads(s)
print(data)
```

```
[{'pages': 475, 'id': '978-1933988177', 'genre':
'technology', 'author': 'Michael McCandless', 'price': 30.5,
'category': ['book', 'paperback'], 'name': 'Lucene in
Action'}, {'pages': 518, 'genre': 'fiction', 'id':
'978-1857995879', 'author': 'Jostein Gaarder', 'price':
3.07, 'category': ['book', 'paperback'], 'name': "Sophie's
World"}]
```

Variable is now a Python list, containing two dictionaries.

# Data Formats: XML

- XML: a **markup** language used to describe data in a structured way using **tags**. Tags are not predefined, but are user defined. Many schemas exist that recommend standardised usage of tags.
- Tags must be opened `<tag>` and closed `</tag>`.
- Tags can contain values as plain text or other nested tags.

```
<note>
  <to>Alice</to>
  <from>John</from>
  <subject>Reminder</subject>
  <body>Remember to buy milk!</body>
</note>
```

Tags are opened and closed

e.g. `<note>...</note>`

We can "nest" tags inside other tags to create a hierarchical structure.

```
<collection>
  <note>
    <to>Alice</to>
    <from>John</from>
    <subject>Reminder</subject>
    <body>Remember to buy milk!</body>
  </note>
  <note>
    <to>John</to>
    <from>Alice</from>
    <subject>Shopping</subject>
    <body>I forgot the milk!</body>
  </note>
</collection>
```

# Data Formats: XML

- Tags can also have zero or more name/value **attribute** pairs, which add extra information to a tag.

XML files often (but not always) contain a header line

Tags are opened and closed  
e.g. `<book>...</book>`

Values can be contained within tags.

Tags can have attributes  
e.g. `currency="eur"`

## File: products.xml

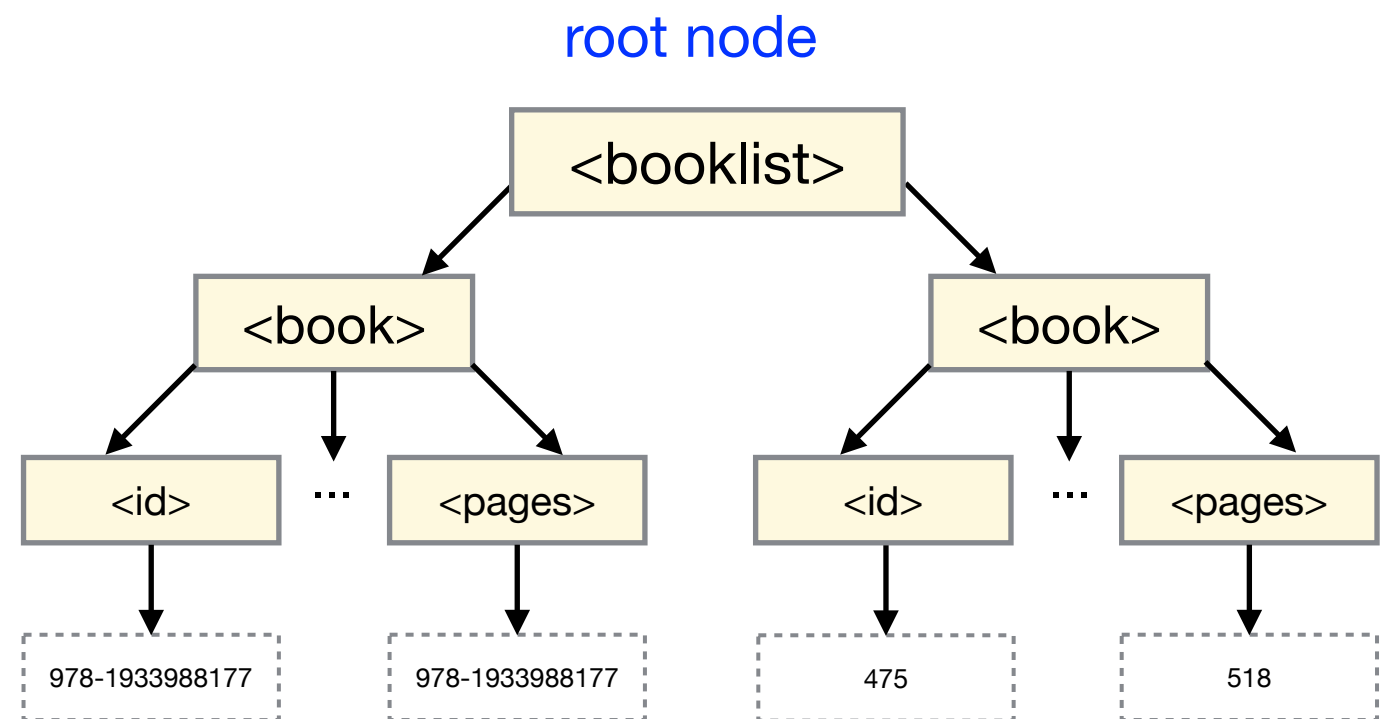
```
<?xml version="1.0" encoding="UTF-8" ?>
<booklist>
  <book>
    <id>978-1933988177</id>
    <category>paperback</category>
    <name>Lucene in Action</name>
    <author>Michael McCandless</author>
    <genre>technology</genre>
    <price currency="eur">30.5</price>
    <pages>475</pages>
  </book>
  <book>
    <id>978-1857995879</id>
    <category>paperback</category>
    <name>Sophie's World</name>
    <author>Jostein Gaarder</author>
    <genre>fiction</genre>
    <price currency="eur">3.07</price>
    <pages>518</pages>
  </book>
</booklist>
```

# Data Formats: XML

- XML is an inherently hierarchical data format, and the most natural way to represent it is with a **tree** with **nodes** at different levels.
- The document itself is the **root node** of the tree. Other nodes are child nodes. If they contain nodes themselves, they are parent nodes. The lowest level of the tree contains **leaf nodes**.

File: products.xml

```
<?xml version="1.0" encoding="UTF-8" ?>
<booklist>
  <book>
    <id>978-1933988177</id>
    <category>paperback</category>
    <name>Lucene in Action</name>
    <author>Michael McCandless</author>
    <genre>technology</genre>
    <price>30.5</price>
    <pages>475</pages>
  </book>
  <book>
    <id>978-1857995879</id>
    <category>paperback</category>
    <name>Sophie's World</name>
    <author>Jostein Gaarder</author>
    <genre>fiction</genre>
    <price>3.07</price>
    <pages>518</pages>
  </book>
</booklist>
```



leaf nodes of this document contain values



# Using XML in Python

- XML is a widely-adopted format. Python includes several built-in modules for parsing XML data.
- The `xml.etree.ElementTree` module can be used to extract data from a simple XML file based on its tree structure.

## File: products.xml

```
<?xml version="1.0" encoding="UTF-8" ?>
<booklist>
  <book>
    <id>978-1933988177</id>
    <category>paperback</category>
    <name>Lucene in Action</name>
    <author>Michael McCandless</author>
    <genre>technology</genre>
    <price>30.5</price>
    <pages>475</pages>
  </book>
  <book>
    <id>978-1857995879</id>
    <category>paperback</category>
    <name>Sophie's World</name>
    <author>Jostein Gaarder</author>
    <genre>fiction</genre>
    <price>3.07</price>
    <pages>518</pages>
  </book>
</booklist>
```

## Import the module and parse the XML file

```
import xml.etree.ElementTree
tree = xml.etree.ElementTree.parse("products.xml")
```

## Find all tags that we are interested in (book) Then find text in nested child tags (name & author)

```
for book in tree.iterfind("book"):
    n = book.findtext("name")
    a = book.findtext("author")
    print("%s by %s" % (n,a) )
```

```
Lucene in Action by Michael McCandless
Sophie's World by Jostein Gaarder
```

# Data Formats: HTML

- HTML is also a markup language similar to XML. Key differences:
  - XML describes data, HTML describes data and presentation.
  - In practice HTML is usually badly-written and invalid.

## File: products.xml

```
<?xml version="1.0" encoding="UTF-8" ?>
<booklist>
  <book>
    <id>978-1933988177</id>
    <category>paperback</category>
    <name>Lucene in Action</name>
    <author>Michael McCandless</author>
    <genre>technology</genre>
    <price>30.5</price>
    <pages>475</pages>
  </book>
  <book>
    <id>978-1857995879</id>
    <category>paperback</category>
    <name>Sophie's World</name>
    <author>Jostein Gaarder</author>
    <genre>fiction</genre>
    <price>3.07</price>
    <pages>518</pages>
  </book>
</booklist>
```

## File: products.html (in browser)

### Book List

#### Lucene in Action

- *Author:* Michael McCandless
- *Genre:* Technology
- *Price:* €30.50
- Paperback, 475 pages

#### Sophie's World

- *Author:* Sophie's World
- *Genre:* Fiction
- *Price:* €3.07
- Paperback, 518 pages

# Data Formats: HTML

- HTML also consists of opening `<tag>` and closing `</tag>`, with other tags and values enclosed within tags.

- Basic page structure:

`<html>...</html>`: Top-level tag

`<head>...</head>`: Page metadata

`<body>...</body>`: Page content

## Book List

### Lucene in Action

- Author:* Michael McCandless
- Genre:* Technology
- Price:* €30.50
- Paperback, 475 pages

### Sophie's World

- Author:* Sophie's World
- Genre:* Fiction
- Price:* €3.07
- Paperback, 518 pages

## File: products.html

```
<html>
  <head>
    <title>Book List</title>
  </head>
  <body>
    <h2>Book List</h2>
    <hr>
    <h3><font color="red">Lucene in Action</font></h3>
    <ul>
      <li><i>Author:</i> Michael McCandless</li>
      <li><i>Genre:</i> Technology</genre></li>
      <li><i>Price:</i> &euro;30.50</li>
      <li>Paperback, 475 pages</li>
    </ul>
    <hr>
    <h3><font color="blue">Sophie's World</font></h3>
    <ul>
      <li><i>Author:</i> Sophie's World</li>
      <li><i>Genre:</i> Fiction</genre></li>
      <li><i>Price:</i> &euro;3.07</li>
      <li>Paperback, 518 pages</li>
    </ul>
    <hr>
  </body>
</html>
```

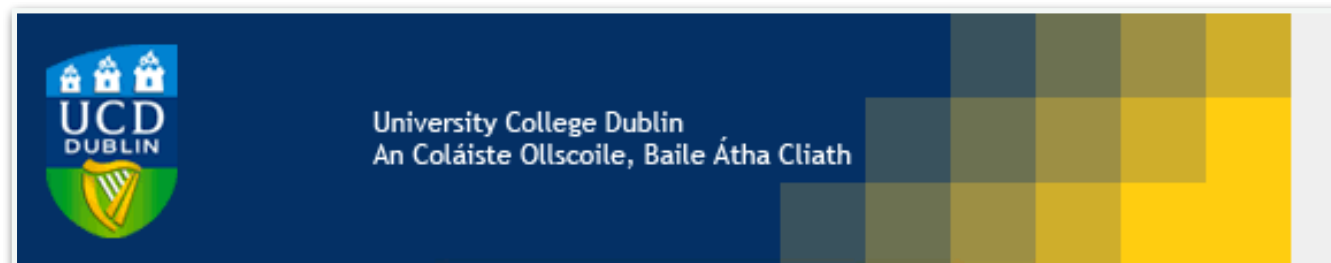
# Web Scraping

---

- **Web scraping:** Technique used to extract data from web sites using a tool that acts as a web browser.
- **Basic steps:**
  1. *Data identification:* Identify and study the structure of the web pages containing the required data.
  2. *Data collection:* Download these web pages in the same way as a web browser does. This may even simulate a user logging in to obtain access.
  3. *Data extraction:* Parse web pages to extract data.
  4. *Data cleaning:* Clean and reformat the data to make it usable.
- **The "rules" of web scraping:**
  - Check a site's terms and conditions before you scrape their pages.
  - Do not hammer a site with too many automated requests.
  - Sites often change their layout, so scrapers often break and need to be re-written.

# Web Data Identification

**Example:** Want to extract list of UCD college names from the page  
<http://mlg.ucd.ie/modules/COMP41680/sample1.html>



## UCD Colleges

UCD College of Arts and Humanities

UCD College of Business

UCD College of Engineering and Architecture

UCD College of Health and Agricultural Sciences

UCD College of Social Sciences and Law

UCD College of Science

## View source: sample1.html

```
<html>
<head>
  <title> UCD Colleges</title>
  <link href="http://www.ucd.ie/stylecss/sub/subpage_dropdown.css"
    rel="stylesheet" type="text/css"/>
</head>
<body>
  <div id="topbar">
    
  </div>

  <div id="main" style="margin: 20px;">
    <h1>UCD Colleges</h1>
    <div id="content">
      <h3><a href="http://www.ucd.ie/artshumanities/">UCD College of Arts and Humanities</a></h3>
      <h3><a href="http://www.ucd.ie/business/">UCD College of Business</a></h3>
      <h3><a href="http://www.ucd.ie/eacollege">UCD College of Engineering and Architecture</a></h3>
      <h3><a href="http://www.ucd.ie/chas/">UCD College of Health and Agricultural Sciences</a></h3>
      <h3><a href="http://www.ucd.ie/socscilaw/">UCD College of Social Sciences and Law</a></h3>
      <h3><a href="http://www.ucd.ie/science">UCD College of Science</a></h3>
    </div>
  </div>
</body>
</html>
```

Inspect the HTML source to  
identify part of the page with  
the relevant data



# Web Data Collection

- The built-in Python `urllib.request` module has functions which help in downloading content from HTTP URLs using minimal code:

```
import urllib.request
link = "http://mlg.ucd.ie/modules/COMP41680/sample1.html"
response = urllib.request.urlopen(link)
html = response.read().decode()
```

Fetch the HTML code  
from the web page

```
lines = html.strip().split("\n")
for l in lines:
    print(l)
```

Split into lines and print  
each line

```
<html>
<head>
  <title> UCD Colleges</title>
  <link href="http://www.ucd.ie/stylecss/sub/subpage_dropdown.css"
    rel="stylesheet" type="text/css"/>
</head>
<body>
  <div id="topbar">
    
  </div>

  <div id="main" style="margin: 20px;">
    <h1>UCD Colleges</h1>
    <div id="content">
      <h3><a href="http://www.ucd.ie/artshumanities/">UCD College of Arts and Humanities</a></h3>
      <h3><a href="http://www.ucd.ie/business/">UCD College of Business</a></h3>
      <h3><a href="http://www.ucd.ie/eacollege">UCD College of Engineering and Architecture</a></h3>
      <h3><a href="http://www.ucd.ie/chas/">UCD College of Health and Agricultural Sciences</a></h3>
      <h3><a href="http://www.ucd.ie/socscilaw/">UCD College of Social Sciences and Law</a></h3>
      <h3><a href="http://www.ucd.ie/science">UCD College of Science</a></h3>
    </div>
  </div>
</body>
</html>
```

# Web Data Extraction

- Many ways to parse HTML pages in Python. The third-party *Beautiful Soup* package is useful for working with badly written HTML pages: <http://www.crummy.com/software/BeautifulSoup>

```
<html>
<head>
  <title> UCD Colleges</title>
  <link href="http://www.ucd.ie/stylecss/sub/subpage_dropdown.css"
    rel="stylesheet" type="text/css"/>
</head>
<body>
  <div id="topbar">
    
  </div>

  <div id="main" style="margin: 20px;">
    <h1>UCD Colleges</h1>
    <div id="content">
      <h3><a href="http://www.ucd.ie/artshumanities/">UCD College of Arts and Humanities</a></h3>
      <h3><a href="http://www.ucd.ie/business/">UCD College of Business</a></h3>
      <h3><a href="http://www.ucd.ie/eacollege">UCD College of Engineering and Architecture</a></h3>
      <h3><a href="http://www.ucd.ie/chas/">UCD College of Health and Agricultural Sciences</a></h3>
      <h3><a href="http://www.ucd.ie/socscilaw/">UCD College of Social Sciences and Law</a></h3>
      <h3><a href="http://www.ucd.ie/science">UCD College of Science</a></h3>
    </div>
  </div>
</body>
</html>
```

In our example, we want to extract the text in the `<h3>` tags.

We can use BeautifulSoup to find all these tags and get the text between them.

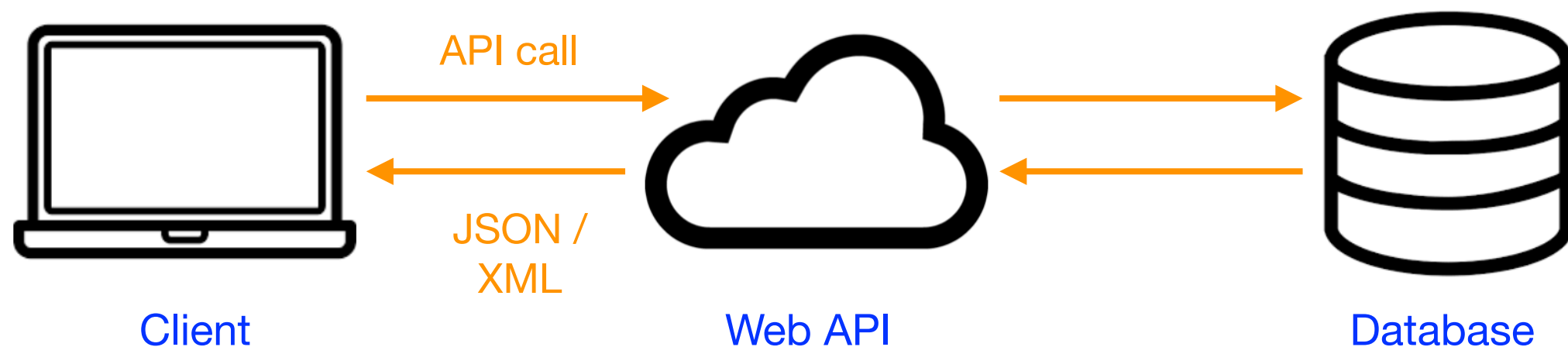
```
import bs4
parser = bs4.BeautifulSoup(html, "html.parser")
for match in parser.find_all("h3"):
    text = match.get_text()
    print(text)
```

```
UCD College of Arts and Humanities
UCD College of Business
UCD College of Engineering and Architecture
UCD College of Health and Agricultural Sciences
UCD College of Social Sciences and Law
UCD College of Science
```

# Web APIs

---

- Instead of manually scraping HTML data, some web sites provide a convenient "official" way of retrieving their data.
- **Web Application Programming Interface (API)**: A web service consisting of a set of HTTP request messages with a definition of the expected structure of the response.
- In contrast to a static website or a fixed dataset (e.g. a single CSV file hosted online), an API can accept a query or call for particular data and respond dynamically with the required result.
- APIs use different data formats to encode calls and responses. Sometimes this can be plain text, but more commonly JSON or XML.



# Web APIs

---

- Web APIs have one or more **endpoints**, which are URLs that can be used to retrieve data - e.g. <https://en.wikipedia.org/w/api.php> is the endpoint for the Wikipedia API.
- Many free Open APIs exist, others are private and paid. Each API has documentation and specifications which determine how information can be transferred and the format of the responses.
- Some APIs have third-party Python packages which provide functions that "wrap" the calls and responses. In other cases, HTTP requests will need to be made and responses parsed manually.
- APIs often have rules and restrictions:
  - Many APIs have rate limits. Can only make limited number of calls from a given IP address/account during a fixed period of time.
  - Some APIs require authentication - i.e. first need to register a user account, and retrieve an authentication token/password.

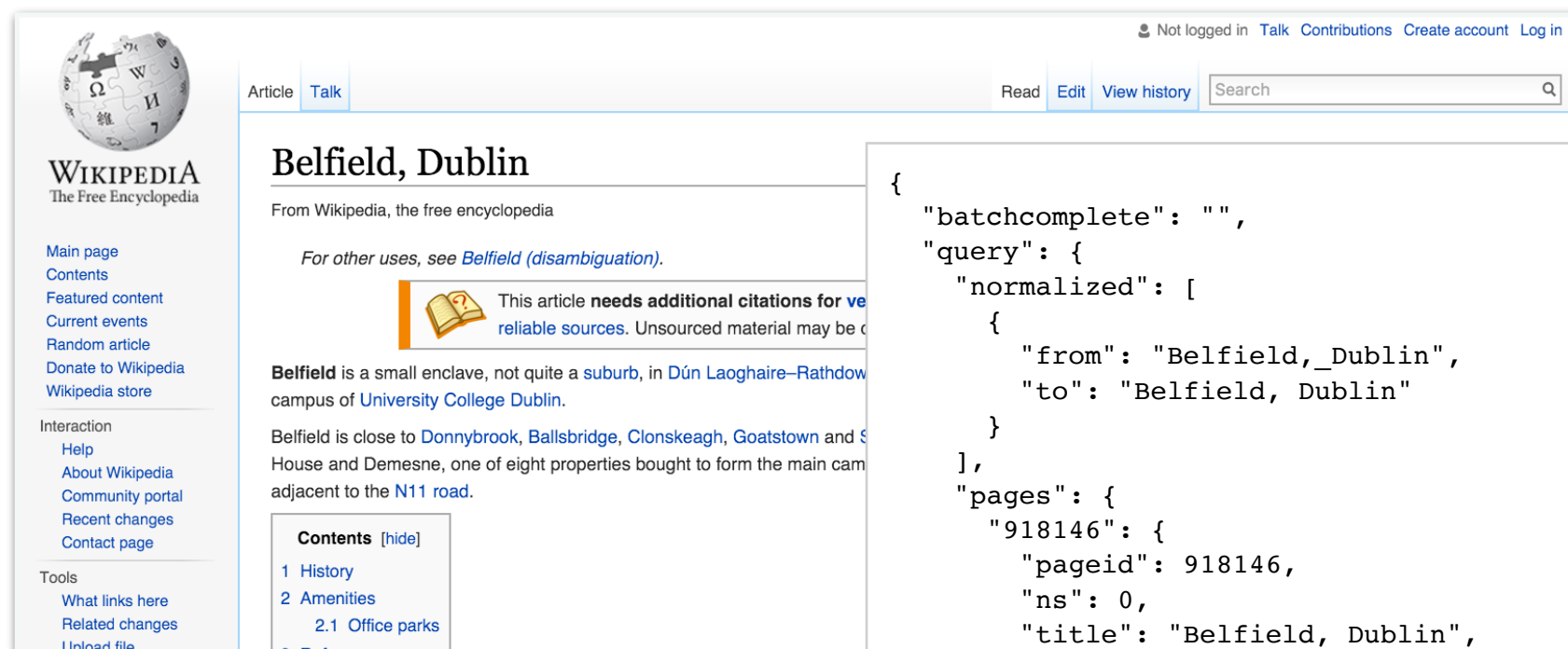
# Web APIs

- **Example:** Wikipedia provides a simple API for retrieving page content in JSON format, endpoint is <https://en.wikipedia.org/w/api.php>

Query URL for the Wikipedia page "Belfield, Dublin"

<https://en.wikipedia.org/w/api.php?>

[format=json&action=query&prop=extracts&exintro=true&titles=Belfield, Dublin](https://en.wikipedia.org/w/api.php?format=json&action=query&prop=extracts&exintro=true&titles=Belfield,_Dublin)



The screenshot shows the Wikipedia article for "Belfield, Dublin". The article title is "Belfield, Dublin" and it is categorized as "From Wikipedia, the free encyclopedia". A notice indicates that the article needs additional citations for reliable sources. The main text describes Belfield as a small enclave, not quite a suburb, in Dún Laoghaire–Rathdown, Ireland, which is synonymous with the main campus of University College Dublin. It is located close to Donnybrook, Ballsbridge, Clonskeagh, Goatstown, and Stillorgan, and takes its name from Belfield House and Demesne, one of eight properties bought to form the main campus of University College Dublin. It is adjacent to the N11 road. The article includes a table of contents with sections for History and Amenities, and a list of office parks.

```
{
  "batchcomplete": "",
  "query": {
    "normalized": [
      {
        "from": "Belfield,_Dublin",
        "to": "Belfield, Dublin"
      }
    ],
    "pages": {
      "918146": {
        "pageid": 918146,
        "ns": 0,
        "title": "Belfield, Dublin",
        "extract": "<p><b>Belfield</b> is a small enclave, not quite a suburb, in Dún Laoghaire–Rathdown, Ireland. It is synonymous with the main campus of University College Dublin.</p>\n<p>Belfield is close to Donnybrook, Ballsbridge, Clonskeagh, Goatstown and Stillorgan and takes its name from Belfield House and Demesne, one of eight properties bought to form the main campus of University College Dublin. It is adjacent to the N11 road.</p>\n<p></p>"
      }
    }
  }
}
```



# Web APIs

- **Example:** The World Bank provides an API for retrieving country-level indicators and facts, endpoint is <http://api.worldbank.org/v2/country>

Query data for Ireland ("ie"): <http://api.worldbank.org/v2/country/ie>

```
<wb:countries xmlns:wb="http://www.worldbank.org" page="1" pages="1" per_page="50" total="1">
  <wb:country id="IRL">
    <wb:iso2Code>IE</wb:iso2Code>
    <wb:name>Ireland</wb:name>
    <wb:region id="ECS" iso2code="Z7">Europe & Central Asia</wb:region>
    <wb:adminregion id="" iso2code=""/>
    <wb:incomeLevel id="HIC" iso2code="XD">High income</wb:incomeLevel>
    <wb:lendingType id="LNX" iso2code="XX">Not classified</wb:lendingType>
    <wb:capitalCity>Dublin</wb:capitalCity>
    <wb:longitude>-6.26749</wb:longitude>
    <wb:latitude>53.3441</wb:latitude>
  </wb:country>
</wb:countries>
```

Query data for Brazil ("br"): <http://api.worldbank.org/v2/country/br>

```
<wb:countries xmlns:wb="http://www.worldbank.org" page="1" pages="1" per_page="50" total="1">
  <wb:country id="BRA">
    <wb:iso2Code>BR</wb:iso2Code>
    <wb:name>Brazil</wb:name>
    <wb:region id="LCN" iso2code="ZJ">Latin America & Caribbean </wb:region>
    <wb:adminregion id="LAC" iso2code="XJ">Latin America & Caribbean (excluding high income)</wb:adminregion>
    <wb:incomeLevel id="UMC" iso2code="XT">Upper middle income</wb:incomeLevel>
    <wb:lendingType id="IBD" iso2code="XF">IBRD</wb:lendingType>
    <wb:capitalCity>Brasilia</wb:capitalCity>
    <wb:longitude>-47.9292</wb:longitude>
    <wb:latitude>-15.7801</wb:latitude>
  </wb:country>
</wb:countries>
```

# Data Serialisation

---

- **Data serialisation**: Process of flattening complex data structures in a programming language, into a format that can be easily stored, transferred, or shared with another program.
- Analogous to a "saved game" in a video game - we save the current state, so that we can restore it at a later time.
- Two common approaches for serialisation in Python are **JSON** and **Pickle**.
- In either case, **serialisation** typically involves transforming the original values into an appropriate format and then writing this serialised data to disk.
- Later we can apply **deserialisation** to reverse the process, typically by loading the serialised data from disk and transforming it back to a copy of the original values.

# Serialisation with JSON

- As we have seen, the `json` module provides a way of converting Python objects into JavaScript Object Notation format, which can be stored and shared.
- We can use `json.dump()` to directly serialise values to a JSON file:
- Similarly, we can use the `json.load()` function to directly deserialise values from a JSON file stored on disk.

Open a file for writing, dump values to disk.

```
import json
ages = {"Steve":25,"Linda":40,"John":33}
fout = open("example.json", "w")
json.dump(ages, fout)
fout.close()
```

Open a file for reading, directly reload a copy of the original values from disk.

```
import json
fin = open("example.json","r")
values = json.load(fin)
fin.close()
print(values)
```

```
{"Linda": 40, "Steve": 25, "John": 33}
```

# Serialisation with JSON

---

- Advantages of using JSON for serialisation:
  - Based on an open standard for data interchange  
<http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf>
  - Language agnostic. Can be read by many other languages and programs.
  - Output is always text-based, can be inspected and verified by humans.
- Disadvantages of JSON serialisation:
  - Does not support all Python data types (e.g. complex objects).
  - It can be slow to read/write large volumes of data.
- If interoperability with other languages/programs is not a requirement, there are other serialisation methods available.

# Serialisation with Pickle

---

- The built-in **pickle** module is the main mechanism provided by Python for serialising data.
- Pickle turns any Python data structure into a bytestream representation, which is easily stored. Then, pickle can be used to reconstruct the original data later.
- Pickle can serialise all of the following:
  - All Python's native datatypes (e.g. integers, strings, floats etc).
  - Lists, tuples, dictionaries, and sets containing any combination of native datatypes or any combination of those data structures.
  - Python functions and custom objects.

Import the module and we can start to serialise objects

```
import pickle
```



# Serialisation with Pickle

---

- Simplest usage of Pickle involves calling the `pickle.dump()` function, which can serialise any Python object to an open file.
- Note that we need to open the file in binary format.

```
import pickle
ages = {"Steve":25,"Linda":40,"John":33}
fout = open("example.pkl", "wb")
pickle.dump(ages, fout)
fout.close()
```

Use the `"wb"` action to open a file for binary writing.

Call `dump()`, specifying the object to serialise and the file object.

- We then reconstruct the original dictionary by calling the `pickle.load()` function, which can deserialise any Python object from an open file.
- Again, we need to open the file in binary format.

```
fin = open("example.pkl", "rb")
x = pickle.load(fin)
fin.close()
```

Use the `"rb"` action to open a file for binary reading.

The dictionary `x` is now identical to the original dictionary `ages`.