

EECS-317 Data Management and Information Processing

Lecture 12 – Data files and Data APIs

Steve Tarzia

Spring 2019

Northwestern

Announcements

- Final project was posted.
 - Part 1 due May 22nd (next Wednesday)
 - Part 2 due June 12th (Wednesday of finals week)
- Another small homework will also be posted, covering MySQL and indexes.

Last lecture: Defining databases and adding data

- Showed how introducing a single identifier column can make foreign keys simpler.
- Looked in detail at an example needing two unique composite keys.
- Gave SQL syntax for creating and altering tables, and modifying data:
 - CREATE TABLE ...
 - INSERT INTO ...
 - DELETE FROM ...
 - UPDATE ...
 - ALTER TABLE ...
- Showed how SQL can be used inside of another language (like Python) to build a database programmatically.

Data files

- A computer **file** is a container for data, and files have:
 - A ***path*** (sequence of folders and a filename):
`C:/Users/Steve/My Documents/my_data.csv`
 - A sequence of data bytes “in” the file (8 bits = 1 byte):
`00010101 10110101 11010101 11010010 10100011 01010101 11110111 ...`
 - Other metadata like *permissions*, depending on the filesystem type.
- Files are:
 - **Persistent**, meaning that they remain in the computer after it is rebooted
 - **Sharable** by other programs running on the computer
- Thus, files allow programs to
 - Save their own data
 - Share data with other programs on the same computer
 - Transfer data between computers
- Databases are a more powerful alternative to plain files (“flat files”), but they are not as *portable*.
 - we still use flat files to exchange bulk data.

Standard data file formats

- The filename *extension* conventionally determines the *file format*.
 - Tells us how to interpret the sequence of bits in the file
- Some file formats use **human-readable** ASCII or UTF-8 **text**.
 - **txt**, **csv**, **json**, **xml**
- More efficient file formats represent data directly in **binary** form.
 - **mat** (matlab), **RData**, **sqlite**, **jpg**, **zip**
- Some files use both formats in two stages:
 - human-readable files that have been compressed to a binary format:
 - **xlsx**, **docx**, **csv.gz**, **txt.gz**

Text encodings

- How do computers store text as ones and zeros?
- Early standard is called the American Standard Code for Information Interchange (ASCII)
 - Developed in the 1960s
 - Uses seven bits per character,
but in practice each character is stored in 8 bits and the top bit is zero.
- ASCII text includes:
 - Lowercase letters, uppercase letters, numbers, punctuation, other symbols
 - Whitespace characters: space, tab, newline, carriage return
 - *Control characters*: null, line feed, vertical tab, bell, escape, delete, backspace, etc.

ASCII TABLE

Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char
0	0	[NULL]	32	20	[SPACE]	64	40	@	96	60	`
1	1	[START OF HEADING]	33	21	!	65	41	A	97	61	a
2	2	[START OF TEXT]	34	22	"	66	42	B	98	62	b
3	3	[END OF TEXT]	35	23	#	67	43	C	99	63	c
4	4	[END OF TRANSMISSION]	36	24	\$	68	44	D	100	64	d
5	5	[ENQUIRY]	37	25	%	69	45	E	101	65	e
6	6	[ACKNOWLEDGE]	38	26	&	70	46	F	102	66	f
7	7	[BELL]	39	27	'	71	47	G	103	67	g
8	8	[BACKSPACE]	40	28	(72	48	H	104	68	h
9	9	[HORIZONTAL TAB]	41	29)	73	49	I	105	69	i
10	A	[LINE FEED]	42	2A	*	74	4A	J	106	6A	j
11	B	[VERTICAL TAB]	43	2B	+	75	4B	K	107	6B	k
12	C	[FORM FEED]	44	2C	,	76	4C	L	108	6C	l
13	D	[CARRIAGE RETURN]	45	2D	-	77	4D	M	109	6D	m
14	E	[SHIFT OUT]	46	2E	.	78	4E	N	110	6E	n
15	F	[SHIFT IN]	47	2F	/	79	4F	O	111	6F	o
16	10	[DATA LINK ESCAPE]	48	30	0	80	50	P	112	70	p
17	11	[DEVICE CONTROL 1]	49	31	1	81	51	Q	113	71	q
18	12	[DEVICE CONTROL 2]	50	32	2	82	52	R	114	72	r
19	13	[DEVICE CONTROL 3]	51	33	3	83	53	S	115	73	s
20	14	[DEVICE CONTROL 4]	52	34	4	84	54	T	116	74	t
21	15	[NEGATIVE ACKNOWLEDGE]	53	35	5	85	55	U	117	75	u
22	16	[SYNCHRONOUS IDLE]	54	36	6	86	56	V	118	76	v
23	17	[ENG OF TRANS. BLOCK]	55	37	7	87	57	W	119	77	w
24	18	[CANCEL]	56	38	8	88	58	X	120	78	x
25	19	[END OF MEDIUM]	57	39	9	89	59	Y	121	79	y
26	1A	[SUBSTITUTE]	58	3A	:	90	5A	Z	122	7A	z
27	1B	[ESCAPE]	59	3B	;	91	5B	[123	7B	{
28	1C	[FILE SEPARATOR]	60	3C	<	92	5C	\	124	7C	
29	1D	[GROUP SEPARATOR]	61	3D	=	93	5D]	125	7D	}
30	1E	[RECORD SEPARATOR]	62	3E	>	94	5E	^	126	7E	~
31	1F	[UNIT SEPARATOR]	63	3F	?	95	5F	_	127	7F	[DEL]

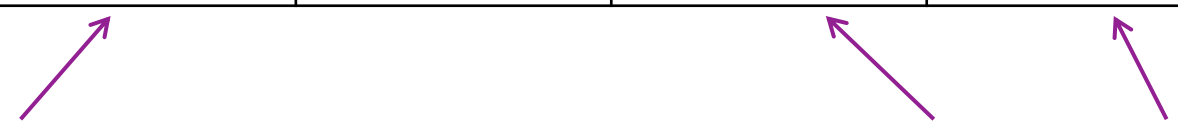
Hexadecimal notation

- Computer programmers often use **hex** notation to represent bit sequences.
- Hex takes four bits and represents them as one of sixteen characters:
 - 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E
- It's the most convenient way for people to represent bit sequences (data):
 - binary 0010 1111 0001 0000 = **0x2F10**
 - “0x” prefix is sometimes added to clarify that what follows is a hexadecimal number.
- ASCII “A” = 0x41 = 01000001 in binary

Decimal	Bits	Hex
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9
10	1010	A
11	1011	B
12	1100	C
13	1101	D
14	1110	E
15	1111	F

“Hello!” in ASCII

	H	e	l	l	o	!
hex	48	65	6C	6C	6F	21
binary	0100 1000	0110 0101	0110 1100	0110 1100	0110 1111	0010 0001



The ASCII table tells us that the letter “H” is represented by this eight-character bit sequence.

“H” **encodes** to 01001000.

01001000 **decodes** to “H”.

The character “l” has the same **encoding** whenever it appears in ASCII text.

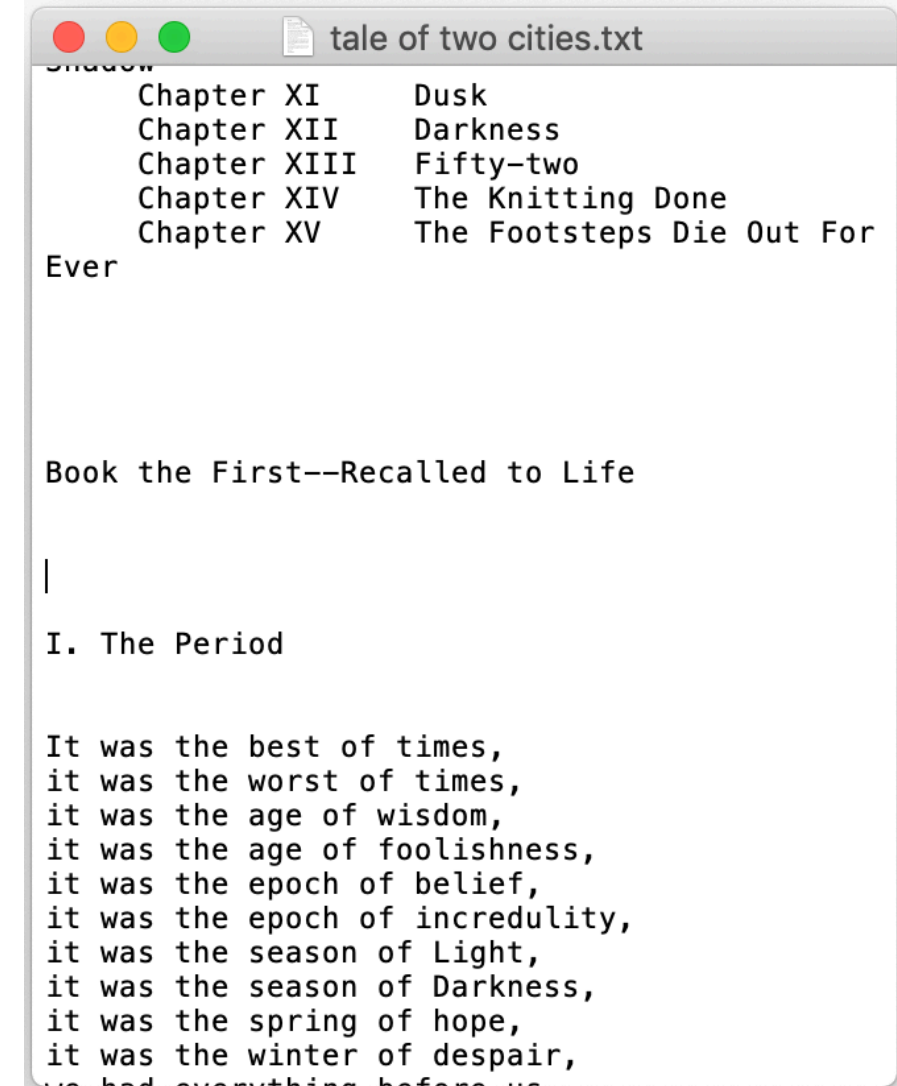
Encoding a text file

C:\Users\Steve\My Documents\tale of two cities.txt

```
42 6f 6f 6b 20 74 68 65 20 46 69 72 73 74 2d 2d |Book the First--|
52 65 63 61 6c 6c 65 64 20 74 6f 20 4c 69 66 65 |Recalled to Life|
0d 0a 0d 0a 0d 0a 0d 0a 0d 0a 49 2e 20 54 68 65 |.....I. The|
20 50 65 72 69 6f 64 0d 0a 0d 0a 0d 0a 49 74 20 |Period.....It|
77 61 73 20 74 68 65 20 62 65 73 74 20 6f 66 20 |was the best of|
74 69 6d 65 73 2c 0d 0a 69 74 20 77 61 73 20 74 |times,..it was t|
68 65 20 77 6f 72 73 74 20 6f 66 20 74 69 6d 65 |he worst of time|
73 2c 0d 0a 69 74 20 77 61 73 20 74 68 65 20 61 |s,..it was the a|
67 65 20 6f 66 20 77 69 73 64 6f 6d 2c 0d 0a 69 |ge of wisdom,..i|
74 20 77 61 73 20 74 68 65 20 61 67 65 20 6f 66 |t was the age of|
20 66 6f 6f 6c 69 73 68 6e 65 73 73 2c 0d 0a 69 |foolishness,..i|
74 20 77 61 73 20 74 68 65 20 65 70 6f 63 68 20 |t was the epoch|
6f 66 20 62 65 6c 69 65 66 2c 0d 0a 69 74 20 77 |of belief,..it w|
61 73 20 74 68 65 20 65 70 6f 63 68 20 6f 66 20 |as the epoch of|
69 6e 63 72 65 64 75 6c 69 74 79 2c 0d 0a 69 74 |incredulity,..it|
```

Data bits in the the file,
shown in hex notation for brevity.
(from “hexdump -C” command)

ASCII or UTF-8 encoding translates each
byte (or up to 4 bytes) to a character



Appearance in text editor

What about other characters we might need?

- ¿Español?, 中文, Ελληνικά
- 😊📠🍟⚽
- Different currency symbols
- Even American English uses “weird punctuation” sometimes.
- A single 8-bit byte will not be enough to store all the possible characters.

UTF-8 to the rescue!

- UTF-8 is now the most common text encoding.
- The latest version includes 136,690 symbols, and more can be added.
 - Can eventually be expanded to more than two million characters
- It's a **variable-length** encoding
 - Characters are represented with one, two, three, or four bytes.
- Backward-compatible with ASCII
 - ASCII text is also valid UTF-8
 - Previous version of Unicode (such as UTF-16) were not widely adopted due to incompatibility with ASCII.

Variable length character encoding with UTF-8

1 st byte	2 nd byte	3 rd byte	4 th byte	# of free bits
0				7 (ASCII)
110	10			11
1110	10	10		16
1111 0	10	10	10	21

- Single-byte characters are identical to ASCII
- First byte tells you how many total bytes to expect
- Every “extra” byte starts with “10”
 - If you start reading in the middle of a character you’ll know it.
 - It’s very easy to know where each new character starts.

Comma Separated Values (CSV)

- CSV is a simple text format for storing tabular data (spreadsheets)
- Each row is represented on one line of text
- Columns are separated by commas
- Values can be enclosed in double quotes ("...") if necessary
 - For example, if value includes comma or newline characters
 - Double quotes within a text value must be “escaped” by using two double quotes
- Values can be empty by having nothing between the commas

NBA_player_of_the_week.csv viewed in Excel

	A	B	C	D	E	F	G	H	I	J
1	PlayerID	TeamID	PositionID	First Name	Last Name	Seasons in Le	Height	Weight	Age	
2	1	20	7	Micheal	Richardson	6	77	189	29	
3	2	14	9	Derek	Smith	2	78	205	23	
4	3	9	2	Calvin	Natt	5	79	220	28	
5	4	15	1	Kareem	Abdul-Jabbar	15	80	225	37	
6	5	2	8	Larry	Bird	5	81	220	28	
7	6	32	9	Darrell	Griffith	4	82	190	26	
8	7	11	7	Sleepy	Floyd	2	83	170	24	
9	8	8	8	Mark	Aguirre	3	84	232	25	
10	9	15	7	Magic	Johnson	5	85	255	25	
11	10	1	8	Dominique	Wilkins	2	86	200	25	
12	11	33	6	Tom	McMillen	9	87	215	32	
13	12	6	9	Michael	Jordan	0	88	215	22	
14	13	7	4	World	Free	9	89	185	31	
15	14	10	7	Isiah	Thomas	3	90	180	23	
16	15	18	6	Terry	Cummings	2	92	220	23	
17	16	6	6	Orlando	Woolridge	3	94	215	25	
18	17	30	1	Jack	Sikma	7	95	230	29	
19	18	22	8	Bernard	King	7	96	205	28	
20	19	25	1	Moses	Malone	8	97	215	29	
21	20	9	8	Alex	English	8	98	190	31	
22	21	26	6	Larry	Nance	3	99	205	26	
23	22	13	1	Herb	Williams	4	101	242	28	
24	23	25	6	Charles	Barkley	1	102	252	23	
25	24	32	8	Adrian	Dantley	9	85	208	30	
26	25	18	9	Sidney	Moncrief	6	89	180	28	

NBA_player_of_the_week.csv viewed as text

PlayerID,TeamID,PositionID,First Name,Last Name,Seasons in League,Height ,Weight,Age

1,20,7,Micheal,Richardson,6,77,189,29

2,14,9,Derek,Smith,2,78,205,23

3,9,2,Calvin,Natt,5,79,220,28

4,15,1,Kareem,Abdul-Jabbar,15,80,225,37

5,2,8,Larry,Bird,5,81,220,28

6,32,9,Darrell,Griffith,4,82,190,26

7,11,7,Sleepy,Floyd,2,83,170,24

8,8,8,Mark,Aguirre,3,84,232,25

9,15,7,Magic,Johnson,5,85,255,25

10,1,8,Dominique,Wilkins,2,86,200,25

11,33,6,Tom,McMillen,9,87,215,32

12,6,9,Michael,Jordan,0,88,215,22

13,7,4,World,Free,9,89,185,31

14,10,7,Isiah,Thomas,3,90,180,23

15,18,6,Terry,Cummings,2,92,220,23

16,6,6,Orlando,Woolridge,3,94,215,25

17,30,1,Jack,Sikma,7,95,230,29

18,22,8,Bernard,King,7,96,205,28

19,25,1,Moses,Malone,8,97,215,29

20,9,8,Alex,English,8,98,190,31

21,26,6,Larry,Nance,3,99,205,26

22,13,1,Herb,Williams,4,101,242,28

23,25,6,Charles,Barkley,1,102,252,23

24,32,8,Adrian,Dantley,9,85,208,30

25,18,9,Sidney,Moncrief,6,89,180,28

26,27,9,Clyde,Drexler,2,95,210,23

27,29,9,Alvin,Robertson,1,98,185,23

28,22,1,Earl,Edwards,4,88,210,25

CSV files represent a single table

- Relational (SQL) models for complex data involve several tables, so you need several CSV files to represent complex data.
- Groups of CSV files are often used for data exchange
- The CSV file can have *column names* in the first row
- However, other important schema information is not stored in CSV:
 - Data types
 - Primary keys
 - Unique key constraints
 - Foreign key relationships
 - Indexes
- The above *metadata* can be included in an SQL script that accompanies the CSV files, or in a human-readable document.
- Each DBMS also has its own proprietary format for exchanging databases, including both the data and metadata.
- SQLite is the simplest. Its just the `*.sqlite` file.

SQL files for data exchange

- **SQL database dumps** are also sometimes used to exchange data.
- These are text files with a listing of all the SQL commands needed to re-generate the database.
 - Includes CREATE TABLE commands and INSERT commands.
 - Running these commands on a fresh/empty database will create a copy of the database that was originally dumped.
- For example:
 - **mysqldump** commandline tool for MySQL
 - **.dump** command in SQLite
 - See the .SQL files in Canvas in the “sqlite databases” folder.
- Disadvantages of SQL as a data exchange format:
 - SQL language dialects differ, so it may not be compatible with all DBMSs
 - It's not very space efficient (lots of SQL syntax is included).

Semi-structured data

- We often must represent complex data in a single file and in a standard way.
- JSON and XML files store semi-structured data
- Not limited to two dimensions like CSV files
- Data is organized in a tree-like/hierarchical way, where any item can have more details below it.
- However, unlike a relational database, there is no clear pre-defined structure or schema for the data.
- **The data defines its own structure.**
- Compared to CSV, it's more difficult to read and is more prone to errors because data elements can be missing.

JSON

- JavaScript Object Notation
- Used in many web applications and data APIs
- Allows an arbitrary amount of **nesting**
- Spaces are ignored, except within quotes.

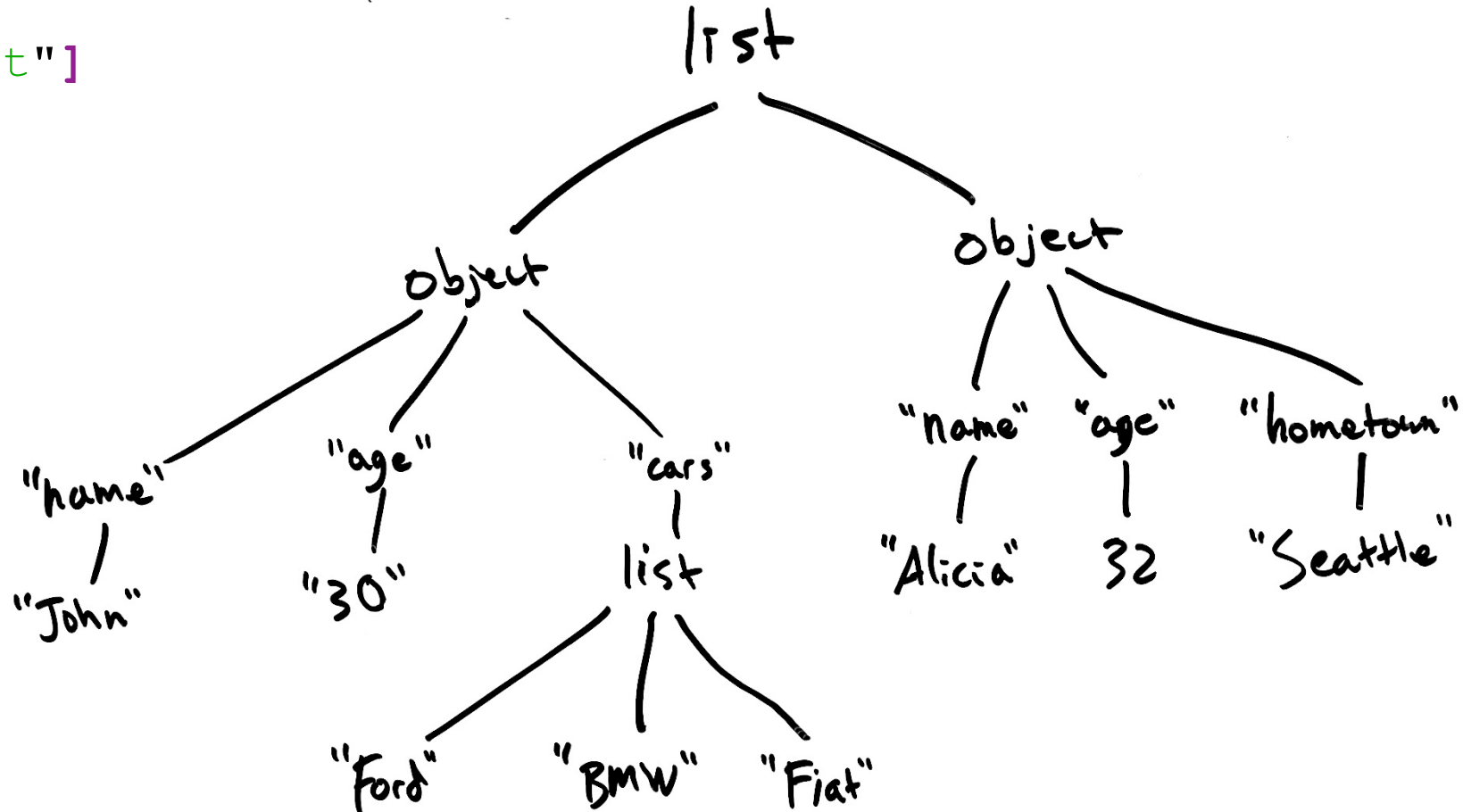
Basic components are:

- **[]** for ordered lists
 - Items are separated by commas
 - Items can be any JSON
- **{ }** for unordered dictionaries/objects
 - Key: value pairs are separated by commas
 - Keys must be strings (text)
 - Values can be any JSON
- Numbers, **true**, **false**, **null**
- Strings (text) in double quotes **"..."**

```
[
  {
    "name": "John",
    "age": 30,
    "cars":
      ["Ford", "BMW", "Fiat"]
  },
  {
    "name": "Alicia",
    "age": 32,
    "hometown": "Seattle"
  }
]
```

JSON data graph example

```
[  
  {  
    "name": "John",  
    "age": 30,  
    "cars":  
      ["Ford", "BMW", "Fiat"]  
  },  
  {  
    "name": "Alicia",  
    "age": 32,  
    "hometown": "Seattle"  
  }  
]
```



XML

- eXtensible Markup Language
- Older than JSON, and now is less common than JSON because many people think XML is unnecessarily complicated.
- HTML is an XML document that defines a web page.

Basic components are:

- Text
- Tags
 - **<tagname>...</tagname>** or just **<tagname>**
 - Have a name, and have XML inside
 - Each start tag has a corresponding end tag, but only if it has data inside.
- Attributes
 - **<tag attr="value" ...>**
 - Appear within tags
 - Attribute name and value must be text
 - Tag can have multiple attributes, but each must have a unique name

```
<people>
  <person name="John"
           age="30">
    <cars>
      <car>Ford</car>
      <car>BMW</car>
      <car>Fiat</car>
    </cars>
  </person>
  <person name="Alicia"
           age="32">
    <hometown city="Seattle">
      </person>
</people>
```

XML data graph example

```
<people>
```

```
  <person name="John"  
    age="30">
```

```
    <cars>
```

```
      <car>Ford</car>
```

```
      <car>BMW</car>
```

```
      <car>Fiat</car>
```

```
    </cars>
```

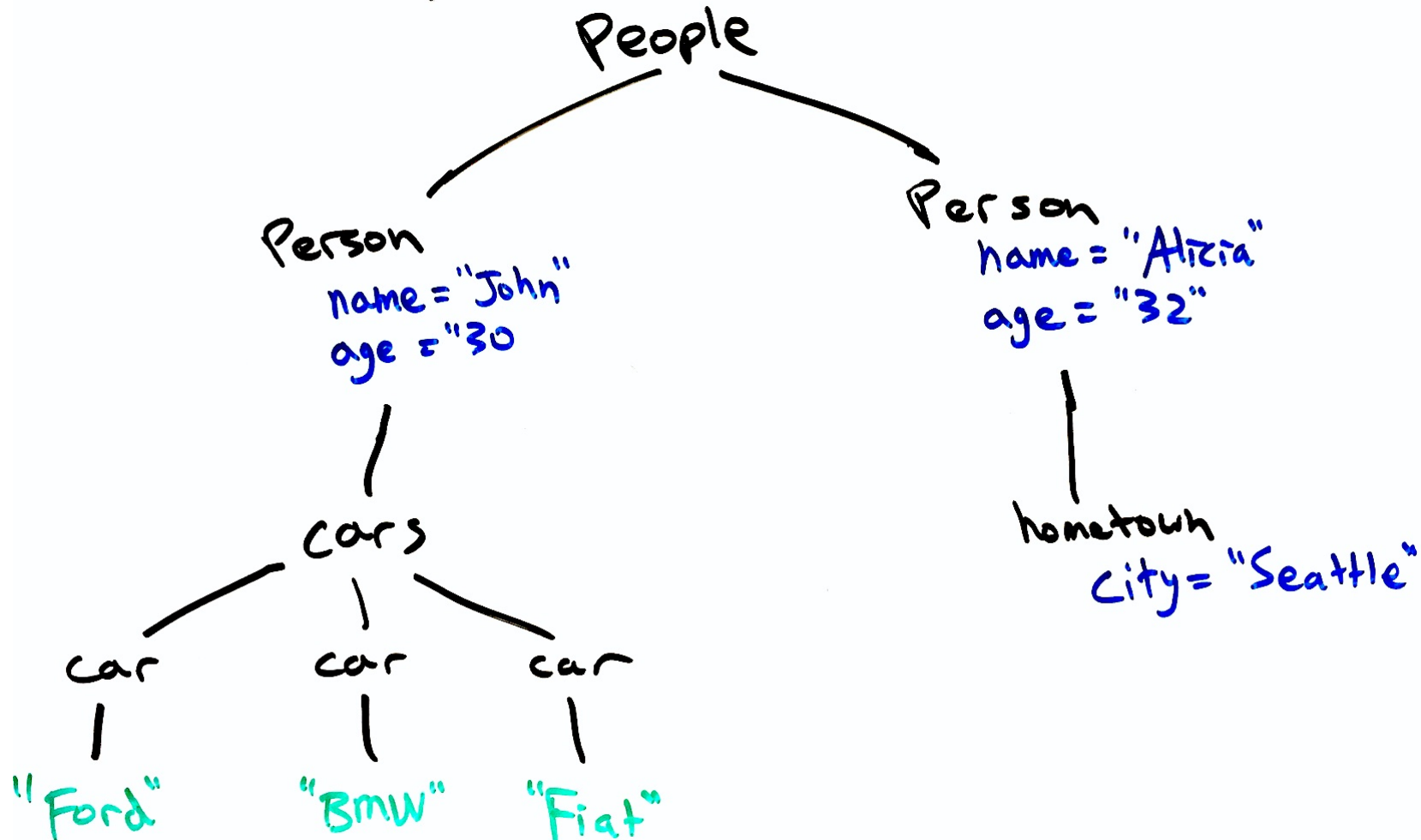
```
  </person>
```

```
  <person name="Alicia"  
    age="32">
```

```
    <hometown  
      city="Seattle">
```

```
    </person>
```

```
</people>
```



Comparison of data exchange formats

	Proprietary	SQL	CSV	JSON	XML
<i>Space efficiency</i>	Compact binary representation	Bloated text with SQL syntax	Text with little extra syntax	Text with little extra syntax	Text with verbose tag names
<i>Compatibility (readable by many)</i>	Must use specific program/DB	Each DBMS has its own SQL dialect	Standardized format	Standardized format	Standardized format
<i>Expressibility (data complexity)</i>	Complex relationships	Complex relationships	Represents a single table	Complex relationships	Complex relationships
<i>Popularity</i>	Rare	Rare	Common	Common	Less common
<i>Flexibility/ rigidity</i>	SQL DBs are have a clearly defined schema that must be obeyed.		Rows all have same columns.	Data and schema are defined together. Different elements can have different attributes.	

- Text-based file formats (SQL, CSV, JSON, XML) are not space efficient, but text files can be compressed using general-purpose file compression utilities like gzip to alleviate the problem (eg., `my_data.json.gz`)

1st half recap

- Data is exchanged by data files (arrays of bits, zeros and ones).
- Several file formats are common:
 - CSV, XML, JSON, and less commonly SQL and proprietary formats.
- Many of these formats are text files with special syntax.
- Text files represent each character with a certain bit sequence.
 - ASCII uses 8 bits (one byte) for each character
 - UTF-8 uses 1-4 bytes for each character, is backward-compatible with ASCII
- CSV files store just one table & can be imported into SQL easily.
- JSON and XML files represent data with complex, nested relationships
 - However, no schema is defined ahead of time.
 - Data itself gives the structure (hence, we call it **semi-structured** data).
 - Python and R scripts can easily load these files.

Bulk vs. online data sources

- So far, we have assumed that we can **bulk export** and **import** data.
 - In other words, we can easily get all the data in one download.
 - Data is exchanged as CSV, JSON, XML, or SQL files:
 - **Dump** file(s) from origin database
 - **Load** file(s) into the destination database
- However, some data sources do not allow bulk access, and instead provide some kind of web-based access to the data:
 - A **data API** may be provided for users to query the data programmatically.
 - Data may be presented in web page for human reading, not intended for programmatic access.

Why is bulk access sometimes not an option?

- If the data set is huge, user many not want to download the entire set.
 - Instead, let the data remain in the **cloud** (on some servers on the Internet), and let users query for their desired data as needed.
- Data may be constantly changing
 - Bulk data files would quickly get “out of date”
 - Instead, provide users some kind of access to a live database.
- Provider may not want general public to have the full data set.
 - For example, Weather Underground lets users get some data, but does not want competing websites to copy all their weather forecasts.

Why isn't SQL used to access cloud data?

- Actually, relational (SQL) databases can be data-sharing platforms.
 - Remember that many users can connect to one SQL database and run queries.
- For example, students in this class accessed a shared MySQL server to access the large Yelp and Stack Overflow databases.
 - *murphy* DB server is on campus, not “in the cloud,” but it could have been.
- A data provider *could* open up its DB servers for public access (without any secret username and password required).
 - In practice, I have **never** seen this done. Why not?
 - DB servers may not be *robust* enough for public access. One poorly written query can slow down the system for everyone.
 - Data users may not know SQL.
 - Database may not support SQL, like MongoDB, DynamoDB, etc.

Data API

- **API** means **A**pplication **P**rogramming **I**nterface
 - It's a very generic term that is applied to different types of *interfaces*:
 - A **software library's API** is the list of *public* functions provided to operate it. In other words, the API defines how your software can use the library.
 - A **web service's API** defines how your software can interact with a remote server to perform various tasks. The API defines what network messages should be passed between the client and server machines.
- **REST APIs** are a type of web service API.
 - REST = REpresentational State Transfer (*bad name!*)
 - A REST API's requests for data look very much like a web browser's requests for html pages and images, so it's familiar and attractive to software engineers.
- REST APIs are now standard way to provide data access to the public.

Hyper Text Transport Protocol (HTTP)

- HTTP is a client-server data exchange protocol
- It was invented for web browsers to fetch pages from web servers
- **Request** specifies:
 - A human-readable header with: *URL*, *method*, (plus some optional headers)
 - An optional *body*, storing raw data (bytes).
- **Response** includes:
 - A human-readable header with *response code*, (plus some optional headers)
 - An optional *body*

Request:

GET /doc/test.html HTTP/1.1

Host: www.test101.com

Accept: image/gif, image/jpeg, */*

Accept-Language: en-us

Accept-Encoding: gzip, deflate

User-Agent: Mozilla/4.0

Content-Length: 35

bookId=12345&author=Tan+Ah+Teck

Request Line

Request Headers

Request
Message
Header

A blank line separates header & body

Request Message Body
(optional for GET)

Response:

HTTP/1.1 200 OK

Date: Sun, 08 Feb xxxx 01:11:12 GMT

Server: Apache/1.3.29 (Win32)

Last-Modified: Sat, 07 Feb xxxx

ETag: "0-23-4024c3a5"

Accept-Ranges: bytes

Content-Length: 35

Connection: close

Content-Type: text/html

<h1>My Home page</h1>

Status Line

Response Headers

Response
Message
Header

A blank line separates header & body

Response Message Body

HTTP methods and responses

Methods

- **GET**: to request a data
- **POST**: to post data to the server, and perhaps get data back, too.

And less commonly:

- **PUT**: to create a new document on the server.
- **DELETE**: to delete a document.
- **HEAD**: like GET, but just return headers

Response codes

- **200 OK**: success
 - **301 Moved Permanently**: redirects to another URL
 - **403 Forbidden**: lack permission
 - **404 Not Found**: URL is bad
 - **500 Internal Server Error**
- ... and many more

A weather information service (REST API)

HTTP Request

```
GET
http://api.wthr.com/[key]/forecast?location=San+Francisco
HTTP/1.1
Accept-Encoding: gzip
Cache-Control: no-cache
Connection: keep-alive
```

HTTP Response

```
HTTP/1.1 200 OK
Content-Length: 2102
Content-Type:
application/json

{ "wind_dir": "NNW",
  "wind_degrees": 346,
  "wind_mph": 22.0,
  "feelslike_f": "66.3",
  "feelslike_c": "19.1",
  "visibility_mi": "10.0",
  "UV": "5", ... }
```

REST API example

Twitter REST API documentation

- <https://developer.twitter.com/en/docs/tweets/post-and-engage/api-reference/post-statuses-update>

Discourse web forum public API documentation:

- <https://docs.discourse.org>

Output examples:

- <https://meta.discourse.org/categories.json>
- <https://meta.discourse.org/latest.json?category=7>
- <https://meta.discourse.org/t/3423.json> (requires authentication)

Inputs and outputs of REST APIs

Request Inputs

- Choice of Method:
 - GET for reading data
 - POST/PUT/DELETE for editing
- URL
 - Usually identifies the type of request, but may also supply parameters:
GET /tweets/**connor4real**
- Query parameters after the main URL
 - Written after a “?” character.
GET /search?**startDate=2018-10-10&search=best+restaurant&api_key=3iur20du9302o3i0d**
- Body
 - Usually form-encoded or JSON

Response Outputs

- Status code
 - 200, 404, 403, etc.
 - Body
 - Usually JSON encoded
-
- Many APIs require that you provide an **API key** or **access token** somewhere your request.
 - This is like a password that identifies you to the service.

REST APIs in Python

- Simplest option is to use [requests](#) library:
- First, “pip install requests”, then:

```
1  import requests, sys
2
3  r = requests.get('https://meta.discourse.org/latest.json')
4  if r.status_code != 200:
5      print("whoops, we got an error response!")
6      sys.exit(-1)
7  response_data = r.json()
8
9  # do whatever we want with the response in Python.
10 # for example, print part of the response
11 for topic in response_data['topic_list']['topics']:
12     print(topic['title'])
13
```

2nd half recap

- Bulk access to data is simple, not always possible
 - Data may be too big, dynamic, or guarded by the owner
- Data is often exposed to users through **data APIs**, which allow users to request pieces of the data. In particular:
 - **REST APIs** use HTTP requests to get data from remote servers.
 - This involves web requests that return JSON data instead of HTML pages.