Data formats & standards

Axel Polleres

March 12, 2019

Data formats

Data

Question.

What is data?

Possible views on data

- Type and scales: numerical (example), categorical (example), or binary (example)
- Text types, e.g.: Emails, tweets, newspaper articles
- Records: user-level data, timestamped event data (e.g.: log files)
- Geo-location data
- Relationship and communication data (Email conversations)
- Sensor data (e.g., streams in Course III)
- Resource types: images, audio, video (not covered in this lecture)
- ...

Data formats

Question.

What is a data format?

What data formats do you know?

What differences between data formats did you encounter?

Data formats

- Data can be stored in many different representations (a.k.a. "formats").
- Some formats are intended to be read and consumed by humans, other by machines
- Further, data is often spread
    - across different *systems & sources*,
    - in different *formats*, and
    - with different *access mechanisms*.

Data formats

Representation of data to encode and to store these data in a computer, and to transfer data between computers

character-encoded data ("text")

- plain (unstructured) text files (*.txt)
- semi-structured data in text files
- structured data in text files

binary-encoded data (0s and 1s)

- images, audio, video
- binary encoding of structured data

Character-encoded, unstructured data

Unstructured, textual data:

- May be hidden inside other formats and needs to be extracted, e.g.:
    - plain-text mails hidden in mailbox archives .mbox format (RFC4155)

  - plain text from a PDF file
  - plain text from a HTML page

Some useful Python libraries:

- pdftotext (for extracting plain text from PDFs)
- beautifulsoup (for extracting data from HTML and XML)

Character-encoded, (semi-)structured data

Question.

What is structured data? What is semi-structured data?

Character-encoded, (semi-)structured data

- Structured data adheres to a particular data definition (a.k.a. "data schema")
  - typically tabular (sets of records as rows of a table)
  - often exchanged using comma-separated values (**CSV**)
- Semi-structured data is structured data that does not (fully) conform to a schema
  - typically represented by (nested) objects or sets of nested attribute-value pairs
  - **exchange formats** for semi-structured data are **XML** and **JSON**
- Graph-structured data
  - a common standard exchange format is: **RDF** /Turtle
- Beware: The boundaries are blurry ... (e.g. there's a JSON serialisation for RDF, called JSON-LD, etc.)

CSV

- Standard defined in RFC 4180
- CSV and its dialects have been in widespread use since the 1960s, but only became standardised in 2005

This is what the RFC 4180 says:

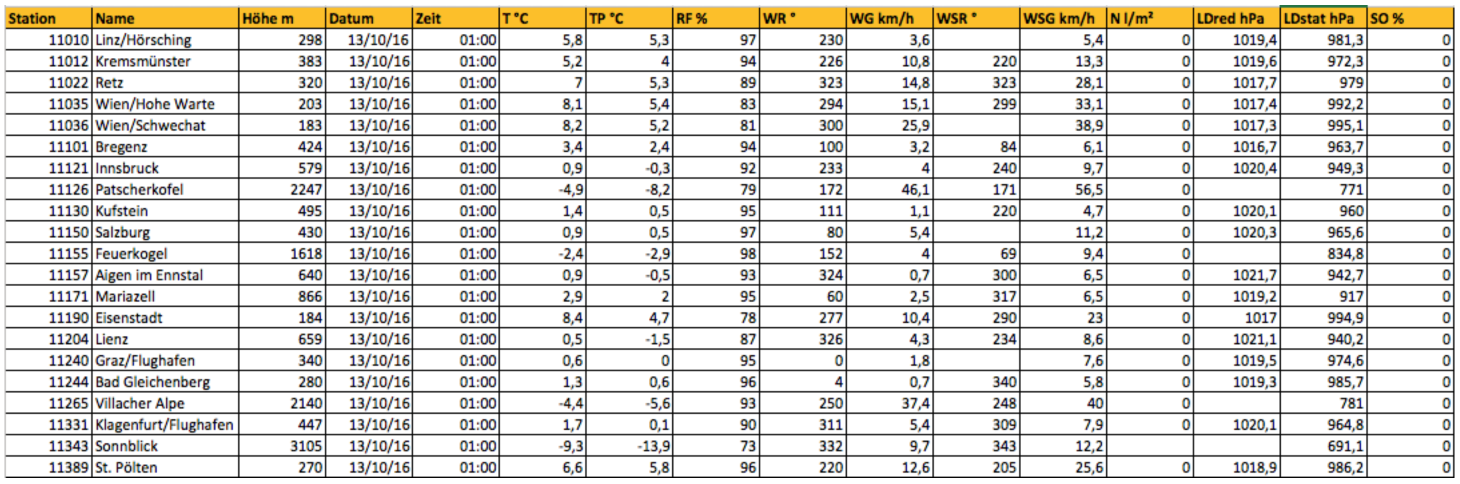
- *"Within the header and each record, there may be one or more fields, separated by commas. Each line should contain the same number of fields throughout the file."*
- *"Each line should contain the same number of fields throughout the file."*
- *"Each field may or may not be enclosed in double quotes"*
- *"Each record is located on a separate line, delimited by a line break (CRLF)."*

Unfortunately, these rules are not always followed "in the wild":

Johann Mitlöhner, Sebastian Neumaier, Jürgen Umbrich, and Axel Polleres. Characteristics of open data CSV files. In 2nd International Conference on Open and Big Data, August 2016.

CSV

- Let's look at an example
- weather data from the Austrian Zentralanstalt für Meteorologie und Geodynamik (ZAMG):



Station	Name	Höhe m	Datum	Zeit	T °C	TP °C	RF %	WR °	WG km/h	WSR °	WSG km/h	N l/m²	LDred hPa	LDstat hPa	SO %
11010	Linz/Hörsching	298	13/10/16	01:00	5,8	5,3	97	230	3,6		5,4	0	1019,4	981,3	0
11012	Kremsmünster	383	13/10/16	01:00	5,2	4	94	226	10,8	220	13,3	0	1019,6	972,3	0
11022	Retz	320	13/10/16	01:00	7	5,3	89	323	14,8	323	28,1	0	1017,7	979	0
11035	Wien/Hohe Warte	203	13/10/16	01:00	8,1	5,4	83	294	15,1	299	33,1	0	1017,4	992,2	0
11036	Wien/Schwechat	183	13/10/16	01:00	8,2	5,2	81	300	25,9		38,9	0	1017,3	995,1	0
11101	Bregenz	424	13/10/16	01:00	3,4	2,4	94	100	3,2	84	6,1	0	1016,7	963,7	0
11121	Innsbruck	579	13/10/16	01:00	0,9	-0,3	92	233	4	240	9,7	0	1020,4	949,3	0
11126	Patscherkofel	2247	13/10/16	01:00	-4,9	-8,2	79	172	46,1	171	56,5	0		771	0
11130	Kufstein	495	13/10/16	01:00	1,4	0,5	95	111	1,1	220	4,7	0	1020,1	960	0
11150	Salzburg	430	13/10/16	01:00	0,9	0,5	97	80	5,4		11,2	0	1020,3	965,6	0
11155	Feuerkogel	1618	13/10/16	01:00	-2,4	-2,9	98	152	4	69	9,4	0		834,8	0
11157	Aigen im Ennstal	640	13/10/16	01:00	0,9	-0,5	93	324	0,7	300	6,5	0	1021,7	942,7	0
11171	Mariazell	866	13/10/16	01:00	2,9	2	95	60	2,5	317	6,5	0	1019,2	917	0
11190	Eisenstadt	184	13/10/16	01:00	8,4	4,7	78	277	10,4	290	23	0	1017	994,9	0
11204	Lienz	659	13/10/16	01:00	0,5	-1,5	87	326	4,3	234	8,6	0	1021,1	940,2	0
11240	Graz/Flughafen	340	13/10/16	01:00	0,6	0	95	0	1,8		7,6	0	1019,5	974,6	0
11244	Bad Gleichenberg	280	13/10/16	01:00	1,3	0,6	96	4	0,7	340	5,8	0	1019,3	985,7	0
11265	Villacher Alpe	2140	13/10/16	01:00	-4,4	-5,6	93	250	37,4	248	40	0		781	0
11331	Klagenfurt/Flughafen	447	13/10/16	01:00	1,7	0,1	90	311	5,4	309	7,9	0	1020,1	964,8	0
11343	Sonnblick	3105	13/10/16	01:00	-9,3	-13,9	73	332	9,7	343	12,2			691,1	0
11389	St. Pölten	270	13/10/16	01:00	6,6	5,8	96	220	12,6	205	25,6	0	1018,9	986,2	0

CSV

You find a CSV version of this data here: `http://www.zamg.ac.at/ogd/`

```
"Station";"Name";"Höhe m";"Datum";"Zeit";"T °C";"TP °C";"RF %";"WR °";"WG km/h";"WSR °";"WSG km/h";"N l/m²";"LDred hPa";"LDstat hPa";"SO %"

11010;"Linz/Hörsching";298;"13-10-2016";"01:00";5,8;5,3;97;230;3,6;;5,4;0;1019,4;981,3;0

11012;"Kremsmünster";383;"13-10-2016";"01:00";5,2;4;94;226;10,8;220;13,3;0;1019,6;972,3;0

11022;"Retz";320;"13-10-2016";"01:00";7;5,3;89;323;14,8;323;28,1;0;1017,7;979;0

11035;"Wien/Hohe Warte";203;"13-10-2016";"01:00";8,1;5,4;83;294;15,1;299;33,1;0;1017,4;992,2;0

11036;"Wien/Schwechat";183;"13-10-2016";"01:00";8,2;5,2;81;300;25,9;;38,9;0;1017,3;995,1;0
```

CSV

Question: What's NOT conformant to RFC 4180 here?

Potential issues:

- different delimiters
- lines with different numbers of elements
- header line present or not

XML

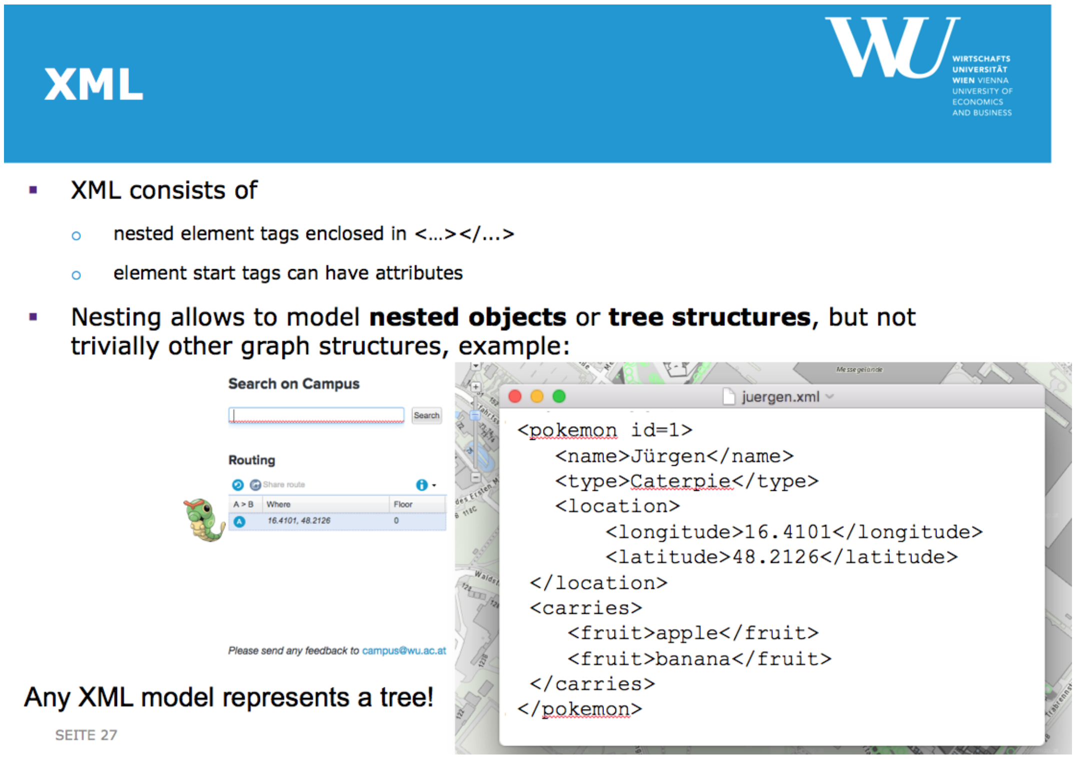
- eXtensible Markup Language, a W3C standard
- Evolved from the Standard Generalized Markup Language (SGML)
- Semi-structured data, with structured portions taking the shape of a *tree* of meta-data (annotation) elements.
  - consisting of **elements**, delimited by named start and end tags `<name>...</name>`
  - with one **root element**
  - arbitrary nesting
  - start tags can additionally carry **attributes**
  - unstructured data are represented as text nodes

Various "companion standards", e.g. schema languages:

- DTD (Document Type Defintion)
- XSD (XML Schema Definition)

XML

Example from the entry tutorial:



XML

- XML consists of
  - nested element tags enclosed in <…></…>
  - element start tags can have attributes
- Nesting allows to model **nested objects** or **tree structures**, but not trivially other graph structures, example:

```
<pokemon id=1>
   <name>Jürgen</name>
   <type>Caterpie</type>
   <location>
       <longitude>16.4101</longitude>
       <latitude>48.2126</latitude>
 </location>
 <carries>
    <fruit>apple</fruit>
    <fruit>banana</fruit>
 </carries>
</pokemon>
```

Any XML model represents a tree!

SEITE 27

XML

Potential issues: e.g.

- broken nesting: non-closed tags
- syntax: missing quotes in attributes, ambiguous meaning of special characters
- check e.g. W3C's markup validator

JSON

- JavaScript Object Notation (JSON)
- an unordered set of key-value pairs of data
  - A JSON Object is enclosed in **{ }**.
  - Keys and values separated by **:**
  - Key value-pairs are separated by **,**
  - JSON objects can be nested, i.e. values can again be objects.
  - arrays (i.e. ordered collections of values) are enclosed in '[' ']'.
- JSON often used in Web applications to transfer JavaScript objects between server-side and client-side application
  - ... JSON has become the most common format for Web APIs
- There is also a schema-definition language for JSON

JSON

Example:

```
{  "id": 10,
    "firstname": "Alice",
    "lastname": "Doe",
    "active": true,
    "shipping_addresses":
    [ { "street": "Wonderland 1", "zip": 4711, "city": "Vienna", "country": "Austria", "home": true },
        { "street": "Welthandelsplatz 1", "zip": 1020, "city": "Vienna", "country": "Austria" },
        { "street": "MickeyMouseStreet10", "zip": 12345, "city": "Entenhausen", "country": "Germany" } ]
}
```

JSON

Example (vs. XML):

```
<customer id="10" active="true">
    <firstname>Alice</firstname>
```

```
      <lastname>Doe</lastname>

      <shipping_addresses>
        <address home = "true"><street>Wonderland 1</street><zip>4711</zip><city>Vienna</city><country>Austria</country></address>
        <address><street>Welthandelsplatz 1</street><zip>1020</zip><city>Vienna</city><country>Austria</country></address>
        <address><street>MickeyMouseStreet10</street><zip>12345</zip><city>Entenhausen</city><country>Germany</country></address>
  </customer>
```

JSON

Jupyter notebooks are represented as JSON documents:

```
 {
  "cells": [
   {
    "cell_type": "markdown",
    "source": [
     "## Assignment 1\n",
     "\n",
     "This assignment is due on mm-dd-YYY-hh:mm by uploading the completed notebook at Learn@WU.\n",
     "\n",
     "### Task 1\n",
     "\n"
```

Summary: Character-encoded, (semi-)structured data

- *Structured data* adhering to a particular schema can be represented *as comma-separated values lists (**CSV**)
- *Semi-structured data* is a form of structured data that does not conform with a fixed schema: (**XML**, **JSON**)
- Outlook: graph-structured data can be represented using standardised data formats (RDF; Turtle; Property Graphs)
- On the one hand, CSV, XML, RDF can be seen as just different serialisation formats of data.
- On the other hand, they employ different characteristic, *abstract* data structures (event-oriented vs. document-oriented encodings).
- There are diffferent tools (and Python libraries!) for dealing with these formats and their variants.

Excursus: Binary encoding of structured data

- For most structured data formats there are also binary (often compressed) binary formats, examples:
  - XML: EXI (Efficient XML Interchange format)
    - compressed, binary XML interchange format
    - a W3C standard recommendation
- JSON: BSON (Binary JSON)
- RDF: HDT (Headers-Dictionary-Triples)
  - a binary, compressed RDF serialization, partially inspired by EXI
  - W3C member submission
  - Co-developed by our institute!

Data Access

Ways to access and get data

Question.

Which access methods can be used to retrieve/download a dataset?

Ways to access and get data

From the Web:

- Download the dataset directly via a URL
- Access the dataset via a API
- *Scraping* the data from a HTML page

Downloading data

Datasets which have an URL (Web address) can be in general directly downloaded

- either manually by pointing the browser to the URL and saving it
- or using programs which access the content and download it to disk or memory

Downloading data

Question.

Can all URLs be easily downloaded? If no, why?

Downloading data

Things to consider when downloading files.

- Some URLs require authentication (we do not cover that case in the lecture)
  - simplest mechanism: .htaccess ,
  - typically more sophisticated methods used nowadays (OAUTH)
- Robots.txt protocol
  - A protocol to guide machines what they are *allowed* to access
  - if existing, located at `http://DOMAIN/robots.txt`
  - Robots/Machines can ignore this protocol.

NOTE: If you want to respect the robots.txt file, you need to first access the file (if available), inspect the allow/disallow rules and apply them to the URL you want to download.

Robots.txt

- Defines which URL sub-directory are *allowed* or *disallowed* to be accessed and by **whom** (User-agent)
- Also allows to recommend a so called crawl-delay (time span between to consecutive accesses)

Robots.txt: Example

`http://data.wu.ac.at/robots.txt`

```
User-agent: *
Disallow: /portalwatch/api/
Disallow: /portalwatch/portal/
Disallow: /portal/dataset/rate/
Disallow: /portal/revision/
Disallow: /portal/dataset/*/history
Disallow: /portal/api/

User-Agent: *
Crawl-Delay: 10
```

In this example, any robot is not allowed to access the specified sub-directories and any robot should wait 10 seconds between two requests

Accessing data via API

Some data sources can be only retrieved via Application Programming Interfaces (APIs).

Question.

Any reasons a data publisher would provide data access via an API rather than providing the data as files?

Accessing data via API

The reason for providing data access via an API:

- the data is generated dynamically/on-demand (*e.g. current temperature*)
- access control (APIs are usually only accessible with access credentials):
  - who is accessing data and from where
  - how often someone can access the data (to avoid overloading the server)
- fine grained access to data
  - all weather information for a certain location vs. downloading GB of global weather data
- easier integration into an existing Application

Accessing data via API: Examples

Last.fm

The Last.fm API allows anyone to build their own programs using Last.fm data, whether they're on the Web, the desktop or mobile devices. Find out more about how you can start exploring the social music playground or just browse the list of methods below.

RottenTomates (Down?)

The Rotten Tomatoes API provides access to Rotten Tomatoes' ratings and reviews, allowing approved companies and individuals to enrich their applications and widgets with Rotten Tomatoes data.

Twitter

The REST APIs provide programmatic access to read and write Twitter data. Author a new Tweet, read author profile and follower data, and more

ProgrammableWeb - an API directory for over 20K Web accessible APIs

Accessing data via a Distributed System API

- Many APIs require authentication and apply a *rate limit* (how many access per time span)
- Specific access methods/protocols (library and protocol)
  - often requires an API key
  - Protocol: There exists different access protocols
    - REpresentational State Transfer (REST): access to an API via HTTP message patterns
    - Simple Object Access Protocol (SOAP) (XML-based)
  - libraries: Typically provide functions that hide the underlying access mechanisms
- Returned data format is typically negotiable (**JSON**, generic or specific_XML_)
- List of Python API's

Accessing data via API: WU BACH API

The WU BACH API *provide machine-readable data of WU's digital ecosystem in line with many OGD [1] initiatives.*

e.g `https://bach.wu.ac.at/z/BachAPI/courses/search?query=data+science`

```
[
   [
      "19S",
      "5585",
      "Data Processing 1",
      [
         [
            6947,
            "Sobernig S."
         ],
         [
            12154,
            "Polleres A."
         ]
      ]
   ]
   /* ... */
]
```

1: Open Government Data

Scraping Web data

Web scraping is the act of taking content from a Web site with the intent of **using it for purposes outside the direct control of the site owner**. [source]

Typical scenarios for Web scraping: Collecting data on

- real estate,
- eCommerce, or
- travel pages

Some examples:

- Get all events from falter.at
- Get visitor statistics for the Austrian Bundesliga

Web scraping also requires to parse a HTML file using dedicated libraries.

WARNING: The legal ground for Web scraping is often not clear and **we do not encourage or suggest to do Web scraping before checking if the site allows it**.
Legal topics around Web scraping will be covered in course III.

Accessing Data: the Python Way

Notebook for Accessing Data: the Python Way

download notebook

The following is the slide version of the notebook

Accessing data sources: Some Python ways

In this part, we cover two data-access methods

1. Loading data from disk
2. Loading data from a Web resource (URL)

We will also learn how to *guess* the file format by inspecting the *metadata* and *the content* of the retrieved data.

Python 3: Opening and closing data streams

The typical steps involved in consuming data are:

1. Open a stream to read the data (either from file or HTTP)
2. Consume the content (e.g. loading the whole content or parts of it)
3. Closing the stream to free up resources:
    - files: allow other processes to access the file (avoid errors/exceptions "*File used by another process*")
    - HTTP: closing a stream allows one to reuse connections

Python 3: Automatically closing data streams

The `with` statement is used to wrap the execution of a block with methods defined by a context manager. This allows common `try` / `catch` / `finally` blocks to be encapsulated for convenient reuse.

Typical use-case : automatically ensure that streams are closed.

Other use cases: timing of functions, printing of logs at the end of a call

```
with COMMAND as C:
  #work with C
```

Loading files from disk

Given that a file is stored on the local machine, we can access the file and inspect or load its content.

There are typically two ways to read the content of a (*text-encoded*) file:

1. Load the whole content of the file and store it in a variable for further processing
2. Read the file line by line (e.g., if files are large)

See also Chapter 7.2 in the Python 3 tutorial

File location

We need the location of the file on disk to load its content.

An **absolute file path** points to the same location in a file system, regardless of the current working directory. To do that, it must include the root directory.

```
Windows: C:\Users\userName\data\course-syllabus.txt
Linux/Mac: /home/userName/data/course-syllabus.txt
```

A **relative path** points to the relative location of a file based on the given/current working directory.

```
Windows:
Linux: ~/data/course-syllabus.txt #starting from home directory
Linux: ../data/course-syllabus.txt #go one folder back, then into data
```

Function: open()

```
help( open )
```

```
Help on built-in function open in module __builtin__:
open(...)
    open(name[, mode[, buffering]]) -> file object

    Open a file using the file() type, returns a file object.
    This is the preferred way to open a file.
    See file.__doc__ for further information.
```

Read content of file into memory

the function `read` consumes the entire contents of the file will be read and returned

```
filePath="./data/course-syllabus.txt"
#open file in read mode
f = open(filePath) # or open(filePath, 'r')

print("Full Output of content:")
content= f.read() # read the whole content and store it in variable content
print(content)
f.close() # do not forget to close the file

#better
with open(filePath) as f: # Carefully with indention and tabs
    content = f.read()
    print(content)
```

see also: ./src/openFile.py

Read content of file into memory

```
Terminal> python3 ./src/openFile.py
b'Full Output of content:\nThis fast-paced class is intended for getting students interested in data science up to speed:\nWe start with an
```

Read a single line from a file

The function `readline` consumes a single line from the file; a newline character (\n) is left at the end of the string

```
filePath="./data/course-syllabus.txt"
#open file in read mode
with open(filePath) as f:# or open(filePath, 'r')
    print("first line: "+f.readline())
    print("second line: "+f.readline())
    print("third line: "+f.readline())
```

Read a single line from a file: Output

```
Terminal> python3 ./src/openFileReadLine.py
b'first line: This fast-paced class is intended for getting students interested in data science up to speed:\n\nsecond line: We start with a
```

see also: ./src/openFileReadLine.py

Read lines from a file using a loop

```
filePath="./data/course-syllabus.txt"
#open file in read mode
with open(filePath)  as f:# or open(filePath, 'r')
    for line in f: # loop over every line in the file (separated by newline)
        print(line)
```

Read lines from a file using a loop: Output

```
Terminal> python3 ./src/openFileLoopLines.py
b'This fast-paced class is intended for getting students interested in data science up to speed:\n\nWe start with an introduction to the fie
```

see also: ./src/openFileLoopLines.py

Resource-saving way to guess the format of a file

Question.

How can we guess the format of a file using as few resources as possible?

Resource-saving way to guess the format of a file

- Inspect the file extension of the file , if availabel (e.g. *.txt*)
- read the first couple of lines, print them and see if you detect any known format syntax patterns (e.g. JSON brackets, CSV delimiters)

Getting the file size of a local file

```
filePath="./data/course-syllabus.txt"

import os
fSize = os.path.getsize(filePath)

print('File size of'+filePath+' is: '+str(fSize) + ' Bytes') # typcasting of an int to str for str concatination
```

fileSize.py: Output

```
Terminal> python3 ./src/fileSize.py
b'File size of./data/course-syllabus.txt is: 435 Bytes\n'
```

see also: ./src/fileSize.py

Loading data from a Web resource (URL)

There exists many libaries in Python 3 to interact with Web resources using the HTTP protocol.

- the urllib library is preinstalled in any Python installation
- the **requests library**, requires to be installed, but is easier to use

HTTP protocol operations

The HTTP protocol is the foundation of data communication for the World Wide Web

The current version of the protocol is HTTP1.1.

A client (browser or library) typically uses the **HTTP GET** operation to retrieve information about and the content of a HTTP URL.

Loading data from a Web resource: urllib

First things first. We need to load the library to be able to use it

```
import urllib.request
```

Afterwards we need to open a connection to the HTTP Server and request the content of the URL

```
urllib.request.urlopen( URL )
```

Urllib: accessing a URL

```
import urllib.request
url="https://bach.wu.ac.at/z/BachAPI/courses/search?query=data+processing"
with urllib.request.urlopen(url) as f:
    print(f.read())
```

Urllib: Output

```
Terminal> python3 ./src/urllib-load.py
b'b"[[\'18W\', \'1629\', u\'Data Processing 1\', [[6947, u\'Sobernig S.\'], [12154, u\'Polleres A.\'], [16682, u\'Hannak A.\']]], [\'19S\',
```

see also: ./src/urllib-load.py

Loading data from a Web resource: requests library

First things first. We need to install and then load the library to be able to use it. the requests library is installed by default on the course container and in any anaconda installation.

```
import requests
```

Afterwards we need to open a connection to the HTTP Server and request the content of the URL

```
requests.get( URL )
```

Requests: accessing a URL

```
import requests
url="https://bach.wu.ac.at/z/BachAPI/courses/search?query=data+processing"

r = requests.get( url )
content=r.text
print(content)
```

Code: Output

```
Terminal> python3 ./src/requests-load.py
b"[['18W', '1629', u'Data Processing 1', [[6947, u'Sobernig S.'], [12154, u'Polleres A.'], [16682, u'Hannak A.']]], ['19S', '5585', u'Data F
```

see also: ./src/requests-load.py

Guessing the file format via the URL

Question.

How can we guess the format of the content of a URL? (...and why would we want to?)

Guessing the file format via the URL

- patterns in the URL
  - file extension: `http://data.wu.ac.at/.../course-syllabus.txt` (**.txt**)
  - query path: `http://..../api/courses`?format=csv (**format=csv**)
- HTTP Response Header
  - contains not only information about the file format

HTTP Response Header

Every HTTP operation has a HTTP request and response header. A HTTP response header is a message from a HTTP server for a request. The header message contains:

- The HTTP status code
- The response header fields
- Empty line
- Message body/content

HTTP Response Header Examples

```
HTTP/1.1 200 OK
Date: Wed, 12 Oct 2016 12:39:12 GMT
Server: Apache/2.4.18 (Ubuntu) mod_wsgi/4.3.0 Python/2.7.12
Last-Modified: Wed, 12 Oct 2016 07:29:32 GMT
ETag: "1b3-53ea5f4498d97"
Accept-Ranges: bytes
Content-Length: 435
Vary: Accept-Encoding
Content-Type: text/plain
```

See also the corresponding RFC. Interesting header fields: **Content-Type** and **Content-Length**

Notice.

Python3 is case-sensitive, meaning that "Content-Type" != "content-type". Sometimes, header fields might be in lower-case or capitalized

Download only the HTTP response header:

Question.

Why would you want to do that?

- HTTP command HEAD

vs.

- HTTP command GET

... Let's check the HTTP specification

HTTP Response Header with Urllib

```
import urllib.request
url="https://bach.wu.ac.at/z/BachAPI/courses/search?query=data+processing"
req =  urllib.request.Request( url , method="HEAD")  # create a HTTP HEAD request
with urllib.request.urlopen(req) as resp:
    header = resp.info()
    # print the full header
    print("Header:")
    print(header)

    ## print the content-type
    print("Content-Type:")
    print(header['Content-Type'])

    ## print the content-type
    print("Content-Length in Bytes:")
    print(header['Content-Length'])
```

Urllib: Output

```
Terminal> python3 ./src/urllib-header.py
b'Header:\nDate: Mon, 11 Mar 2019 22:17:44 GMT\nServer: Zope/(2.13.23, python 2.7.9, linux2) ZServer/1.1\nSet-Cookie: BACH_PRXY_ID=XIbeiHTv6
```

see also: ./src/urllib-header.py

HTTP Response Header with Requests

```
import requests
url="https://bach.wu.ac.at/z/BachAPI/courses/search?query=data+processing"

r = requests.head( url ) # would also work with a HTTP Get
headerDict=r.headers
print(headerDict)
```

Requests: Output

```
Terminal> python3 ./src/requests-header.py
b"{'Date': 'Mon, 11 Mar 2019 22:17:44 GMT', 'Server': 'Zope/(2.13.23, python 2.7.9, linux2) ZServer/1.1', 'Set-Cookie': 'BACH_PRXY_ID=XIbeiK
```

see also: ./src/requests-header.py

Inspect Request library HTTP Response Headers

```
import requests
url="http://datascience.ai.wu.ac.at/ss17_datascience/data/course-syllabus.txt"

r = requests.head( url ) # would also work with a HTTP Get
headerDict=r.headers

#>
#print all available response header keys
print("Header")
print(headerDict)

#access content-type header
if "Content-Type:" in headerDict:
    print("Content-Type:")
    print( headerDict['Content-Type'] )
```

Requests: Output

```
Terminal> python3 ./src/requests-header-inspect.py
b"Header\n{'Date': 'Mon, 11 Mar 2019 22:17:44 GMT', 'Server': 'Apache', 'Location': 'https://datascience.ai.wu.ac.at/ss17_datascience/data/c
```

see also: ./src/requests-header-inspect.py

Handling data formats and character encodings

Character Encodings

Question.

Why is that occurring? "J�rgen"

Character Encodings

- Textual symbols (=characters) are encoded in bits and bytes differently, depending on the number of characters in the overall symbol set:
  - ASCII needs only 1 byte for its 127 characters.
  - Unicode uses up to 4 bytes (the common UTF-8 encoding uses variable length of 1-4 bytes)
  - Assuming a wrong encoding when reading a textfile, or using software tools that cannot handle the input encoding correctly produces arifacts like the one in the last slide.

Further notebooks

- Encoding and reading textfiles
- Dealing with CSV
- Dealing with JSON
- Dealing with RDF
- Dealing with XML

Excursus: How and where to find data?

Question

Question.

How can you find interesting datasets for your project?

Possible ways to find data

- Search online using Google, Bing, Yahoo
- Follow questions/search on Quora, Stackoverflow,...
- Blogs about datascience
- Curated lists of datasets
- Twitter ('#dataset' '#opendata')
- Data portals

Google, Bing, Yahoo

Many datasets can be found using a Web Search engine such as Google, Bing or Yahoo

Combine your keyword search with tokens such as "csv", ".csv".

Such search engines offer also more advanced search features to filter for particular data formats

- Google: *filetype:csv*, *filetype:json* (see file types indexed by google)
- Yahoo: *fileformat:csv*
- Bing: *filetype:csv*, *ext:csv* (Bing search features)

Fileformat search on Bing and Yahoo does not return very good results.

- New in Google: Dataset Search

(Try out e.g.: 'WU Vienna Lectures')

- Our 'homebrewn' Open Dataset search (Spatio-temporal search in Open Data)

Follow questions/search on Quora, Stackoverflow, Reddit

Quora and Stackoverflow are **question-and-answer** sites where people can pose any question and receive community answers

Some direct links:

- Quora: tag datasets (a list of questions tagged with *dataset*)
- Quora: Where-can-I-find-large-datasets-open-to-the-public
- Stackoverflow: tag dataset
- Reddit.com group datasets

Notice.

The popular useful platform change over time. Hint: Follow Metcalfe's Law

Blogs about datascience

Some *general datascience* blogs regularly have posts about datasets

- Dataquest.io Blog
- thedataincubator.com blog

Lists of datascience blogs:

- 100 Active Blogs on Analytics, Big Data, Data Mining, Data Science, Machine Learning
- another currated lists of datascience blogs
- Quora: What are the best datascience blogs?

Curated lists of datasets

Many people also provide a curated lists of public datasets or APIs to datasets. These lists can be typically found via a Google/Bing/Yahoo Search

Some examples:

- Git repository of Fivethirtyeight
- kaggle.com dataset market
- awesome-public-datasets (Github)

Open Data Portals

So called (Open Data) portals are catalogs for datasets.

- Austrian Open Government Data portal data.gv.at
- Austrian Open Data portal opendataportal.at
- Vienna Open Data open.wien.gv.at
- WU Open Data portal data.wu.ac.at
- European Data portal europeandataportal.eu/

Further links:

- The WU project, Open Data Portal Watch maintains a list of 261 Open Data portals

Question

Question.

What should you consider if you use *"public"* datasets

Consuming *"public"* datasets

Public does not necessarily mean **free**

Many public datasets come with certain restrictions of what one is allowed to do with the data.

The data license (if available) typically specifies the following questions:

- Is it allowed to reuse the data in my project/application?
- Is it allowed to merge the dataset with other datasets?
  - If it is allowed, with what other licenses can the data be merged
- Is it allowed to modify the dataset (e.g. remove or transform values)?

Notice.

More about licenses of datasets in SBWL3