Data Formats

Reading / References

- UTF FAQ[1]
- Brief explanation of ASCII and Unicode encodings[2]
- ASCII Table[3]
- Unicode Map[4]
- Python's Unicode support[5]
- JSON's description[6]
- Python's JSON support[7]
- "Beginning JSON" book[8] chapters 4 through 6. Focuses on usage in Javascript.
- Twitter's Rest API[9]
- Character Encodings Wikipedia page[10]
- Wikipedia page on Unicode[11]
- Wikipedia page on UTF-8[12]

Practice questions on the reading

- What does ASCII stand for? When was it introduced?
- How many different characters can be represented in the ASCII encoding?
- Why is ASCII not sufficient for handling text data?
- How many different characters can the utf-8 encoding represent?
- What do the BE and LE variants stand for? What is the problem of "endianess"?
- How many bytes would the string "hello" take to represent in utf-8? What about utf-16BE and utf-32BE?
- What is the default string format in Python?
- How do we represent a particular Unicode character by an escape sequence?
- How do we turn a Python value of type "bytes" (the kind that we obtain from a web request) into a string?
- Can a JSON object be "empty"?
- What number representations does JSON allow?
- Does JSON allow boolean values?
- True or False: All whitespace in a JSON document is ignored
- Is a web page an XML document?

[1] http://unicode.org/faq/utf_bom.html
[2] http://net-informations.com/q/faq/encoding.html
[3] http://www.asciitable.com/
[4] http://www.unicodemap.org/
[5] https://docs.python.org/3/howto/unicode.html
[6] http://json.org/
[7] https://docs.python.org/3/library/json.html
[8] https://www.safaribooksonline.com/library/view/beginning-json/9781484202029/
[9] https://dev.twitter.com/rest/public
[10] https://en.wikipedia.org/wiki/Character_encoding
[11] https://en.wikipedia.org/wiki/Unicode
[12] https://en.wikipedia.org/wiki/UTF-8

Notes

When trying to read or write data, there are a number of considerations:

- **Communication** with the data source or target. This typically involves some established data transmission protocol, for instance HTTP which we will touch on later. It may also include questions of authentication and authorization.
  - **Authentication** establishes that the requester is who they say they are.
  - **Authorization** establishes whether the requester has the necessary credentials to access the given resource.
- The **encoding** for the communication of the data. The data we want to transfer is almost invariably text of some sort, but what we transmit is binary information. There are numerous encodings that turn this binary information into text and back, and even though some are established standards, one needs to pay attention to the individual encoding in question for their particular case.
- The **format** of the data. This is a detailed description of how the textual information that was transmitted is to be interpreted. In other words, how are we to turn this string of text into a meaningful object? We will spend some time looking at the most common formats.

We will discuss the question of communication later on. For now we will briefly discuss encodings, and will spend most of our time discussing formats.

Character Encoding

Character encodings allow us to associate characters to numbers that can then be stored in a computer.

One of the earliest standards in that regard is **ASCII**, which represents 128 different characters with 7 bits. That space is enough to fit both lowercase and uppercase letters of the alphabet, as well as numbers and most common punctuation marks. It also includes a number of common "control codes" to describe key operations, like "carriage return" and "new line" for starting a new line, or a control code for "backspace". Effectively every time you type something in your keyboard, it gets converted to a corresponding ASCII character.

In Python, you can use the functions `ord` and `chr` to go back and forth between an ASCII character and its corresponding number. For example, the character `'a'` corresponds to the number 97, often written in hexadecimal form as `0x61`:

```
ord('a')                # <-- 97
0x61                    # <-- 97
chr(97)                 # <-- 'a'
chr(ord('a') + 3)       # <-- 'd'
for i in range(0,26):
  print(chr(97 + i))
```

Even though this encoding standardized matters for a little while, it was very limited as computers spread around the world. There was a need to accomodate various languages and alphabets, and the 128 characters of ASCII were terribly insufficient. A number of encodings evolved at this time, each extending the ASCII encoding in some way, many simply by the addition of one more bit for a total of 256 different characters and 128 new characters, enough to accomodate the alphabet of a new language. If you search for various encodings online, you will see the proliferation of possible encoding formats. This was clearly a nightmare in terms of communication. The same file when opened in a computer in France might look very different compared to the same file opened in a computer in Germany under a different encoding.

Around 1988 efforts to provide a unified access to all the world's alphabets started, and evolved into what we now know as Unicode and its various encodings. The **Unicode** standard describes "codes" (numbers) for over 128,000 characters, including most of the world's alphabets as well as numerous special symbols from currency to emojis. The space has room for more characters in the future, and various committees are working on doing just that (see for instance Unicode emoji[13]).

Practice The unicode space uses 21 bits. How many different characters can it represent?

These "codes" can then be mapped in a variety of ways into actual bytes, which can then be transmitted between computers. This is achieved in a variety of ways, referred to as "Unicode Transfer Formats". The most common of these format is called UTF-8. Some if its key features are:

- It uses only 1 byte to represent all ASCII characters. In that sense it is a strict extension of ASCII and can accomodate all ASCII documents without change.
- It uses 2 bytes for most common language alphabets, covering 1920 different characters and accomodating most common usages.
- It uses 3 bytes for some of the more extended alphabets, including all Chinese, Japanese and Korean characters.
- 4 bytes are used for other symbols, for instance numerous mathematical symbols, emojis, other notations.

UTF-8 is by far the most commonly used encoding. Almost every resource you find will be in that format, including over 90% of webpage content. But it is important to be aware of the alternatives.

In Python, chr (or unichr on version 2), can return the unicode character represented by a given number:

```
chr(0x2194)    # <-- returns a double-sided arrow symbol.
```

In general, in Python 3 all strings are automatically considered to be in unicode encoding, more specifically UTF-8. But when reading from a file, it is still often required that you specify the encoding to be used.

[13] http://unicode.org/emoji/charts/full-emoji-list.html

Data Formats

The resulting textual data might be in any number of different formats. Being able to read and process all (most) of these formats is paramount, and most programming languages offer ready access via numerous libraries. Here is just a small collection of the different formats that your data might come in.

Binary While we will discuss mostly text formats, it is worth mentioning that, especially in the past, data was often communicated via an ad-hoc, often proprietary, format that is rarely human-readable. This was done partly to save space, but the practice is somewhat more rare nowadays. Most image data is still communicated in this form however. Try to open an image file in your favorite text editor one day. Excel files used to follow such a binary format until recent years, when an XML format started being in use (the xlsx extension).

Text Oftentimes our "data" is purely textual, and calling on further analysis. For example they might be a book or poem, a contract, transcripts of a conversation etc. Textual data can appear in any number of formats, from a Word document (docx format) to a plain text file (txt) to the Markdown and RestructuredText formats that allow some semantic markup of the plain text. We will probably not look much into these.

CSV A popular tabular data format is the so-called "comma-separated-format". Information is written in rows, with individual entries in a row seperated by a comma. Double quotes are typically used to identify strings and to allow a comma to appear in the middle of an entry. Tabular data in Excel can be exported in this form.

HTML Quite often our data is actually the information in a web page document. We'll learn ways to delve into such a document and extract key pieces of information.

XML A fairly popular structured data format where tags are used, similar to HTML, to convey information.

JSON A popular loosely-structured data transmision format, inspired by Javascript's syntax for objects. It is extensively used these days, and we will now delve more into it.

JSON

We will focus on JSON and XML, as these are two of the most popular transmission formats. They are both used to represented hierarchical structures. JSON stands for "JavaScript Object Notation", and it is fairly simple. Here is an example JSON document returned from a search query to Twitter:


```
{
   "search_metadata": {
      "completed_in": 0.032,
      "count": 15,
      "max_id": 773149726313099268,
      "max_id_str": "773149726313099268",
      "next_results": "?max_id=771520407623139328&q=%40HanoverCollege&include_entities=1",
      "query": "%40HanoverCollege",
```

```
      "refresh_url": "?since_id=773149726313099268&q=%40HanoverCollege&include_entities=1",
      "since_id": 0,
      "since_id_str": "0"
   },
   "statuses": [
      {
         "contributors": null,
         "coordinates": null,
         "created_at": "Tue Sep 06 13:23:53 +0000 2016",
         "entities": {
            "hashtags": [],
            "symbols": [],
            ...
         },
         ...
      },
      ...
   ]
}
```


Briefly:

- A JSON document consists of an "object", indicated by an open-close pair of curly braces.
- The object consists of a series of key-value pairs, where the key is some string in quotes, and the value can be:
- the keyword null
- a number
- a string
- an array indicated by square brackets, whose elements are in turn any of the types listed here
- another object

In the above example, we see that the top level object contains two entries, with keys "search_metadata" and "statuses". The former's value is an object, and the latter's value is an array.

And that's it! The format is intentionally kept simple. For an application to make use of such information, it would need to know what the possible keys are and what the corresponding values mean. This is often described as the API. Twitter's API describes all these entries in detail. It is worth a look. We will examine it closely during our first lab assignment.

Those familiar with Javascript will recognize the above syntax, as we can create literal objects that way. And this is correct. JSON is however a bit more strict about the format.

Practice: Write a JSON object that contains the following:

- Your email login.
- Your first and last name.

- Your year in college.
- A list of the courses you are enrolled in, each course containing the information of its department code, its number, its name, and your grade.

XML

XML, standing for eXtensible Markup Language, was a major effort to standardize data description and transmission with a format similar to HTML.

- Open/Close tags are used to denote a nesting hierarchy.
- The tag names are standardized to indicate specific meaning.
- Attributes to each tag can enrich the provided information.

Here is a sample XML file, taken from here[14]:

```
<?xml version="1.0"?>
<catalog>
   <book id="bk101">
      <author>Gambardella, Matthew</author>
      <title>XML Developer's Guide</title>
      <genre>Computer</genre>
      <price>44.95</price>
      <publish_date>2000-10-01</publish_date>
      <description>An in-depth look at creating applications
      with XML.</description>
   </book>
   <book id="bk102">
      <author>Ralls, Kim</author>
      <title>Midnight Rain</title>
      <genre>Fantasy</genre>
      <price>5.95</price>
      <publish_date>2000-12-16</publish_date>
      <description>A former architect battles corporate zombies,
      an evil sorceress, and her own childhood to become queen
      of the world.</description>
   </book>
   <book id="bk103">
      <author>Corets, Eva</author>
      <title>Maeve Ascendant</title>
      <genre>Fantasy</genre>
      <price>5.95</price>
      <publish_date>2000-11-17</publish_date>
      <description>After the collapse of a nanotechnology
      society in England, the young survivors lay the
      foundation for a new society.</description>
   </book>
   <book id="bk104">
      <author>Corets, Eva</author>
      <title>Oberon's Legacy</title>
      <genre>Fantasy</genre>
      <price>5.95</price>
```

[14] https://msdn.microsoft.com/en-us/library/ms762271(v=vs.85).aspx

```
        <publish_date>2001-03-10</publish_date>
        <description>In post-apocalypse England, the mysterious
        agent known only as Oberon helps to create a new life
        for the inhabitants of London. Sequel to Maeve
        Ascendant.</description>
    </book>
    ...
</catalog>
```

There are various documents describing what tags to use depending on the situation, and there are many standards that focus on particular usage. This is a topic we may revisit later. For now note that XML is somewhat more "verbose" than JSON, with the opening and closing of all tags. It also does not readily provide arrays, though the contents of a tag are in effect a sort of list. But at the same time it offers a more diverse way of associating tags/names to bits of information. For example the `id="..."` attributes used above are at a different level than the `author` tags below, they are literally a part of the opening tag. Under different conventions we could have written those instead as children tags like `<id>bk104</id>`.

Practice: Write an XML that would represent the same information you wrote earlier for JSON.