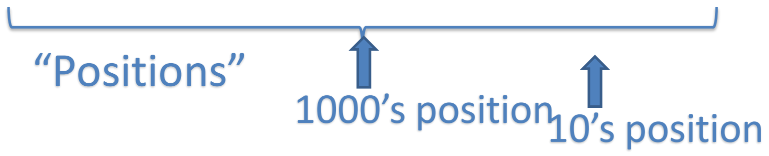
Character Encoding

With a smattering of coverage of number systems - Decimal, Hexadecimal and Binary (just a little…)

Base 10 Number System (i.e., decimal)

How is the base-10 number 128 “constructed” ? For example, what combination of powers of “10” does it take to equal 128?

Power	10^6	10^5	10^4	10^3	10^2	10^1	10^0
Value Range	0d {0-9}	0d {0-9}	0d {0-9}	0d {0-9}	0d {0-9}	0d {0-9}	0d {0-9}
Decimal Equiv for value of 1	1,000,000	100,000	10,000	1,000	100	10	1



Base 2 Number System (i.e., binary)

How is the base-2 number 128 “constructed” ? For example, what combination of powers of “2” does it take to equal 128?

Octet / Byte

Nibble

Power	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
Value Range	0b{0-1}	0b{0-1}	0b{0-1}	0b{0-1}	0b{0-1}	0b{0-1}	0b{0-1}	0b{0-1}
Decimal Equiv for value of 0b1	128	64	32	16	8	4	2	1

“Positions”

8’s position

Base 16 Number System (i.e., hexadecimal)

How is the base-16 number 128 “constructed” ? For example, what combination of powers of “16” does it take to equal 128?

Power	16^3	16^2	16^1	16^0
Value Range	0x {0-F} 0d {0-15}	0x {0-F} 0d {0-15}	0x {0-F} 0d {0-15}	0x {0-F} 0d {0-15}
Decimal Equivalent for value 1	4096	256	16	1

Hex	Dec
0-9	0-9
A	10
B	11
C	12
D	13
E	14
F	15

“Positions”

16’s position

Binary Nibbles and Hexadecimal digits – what gives?

- Since 16 is a power of 2, the hex numbering system is the most often used base numbering system for shorthand notation of binary
- Each "nibble" (4 binary bits) can range in value between 0b0000 and 0b1111
  - 0b1111 is what in decimal?
  - what is the value range of a single hex digit?
- Since each digit of a hexadecimal number can also range between decimal 0 and 15 a single hex digit can ALWAYS be used to represent a 4 binary bits

Numbering System Example

- What are the hexadecimal and binary equivalent of each of the following decimal numbers

DECIMAL	BINARY	HEXADECIMAL
4097		
27		
284		

Powers of 2

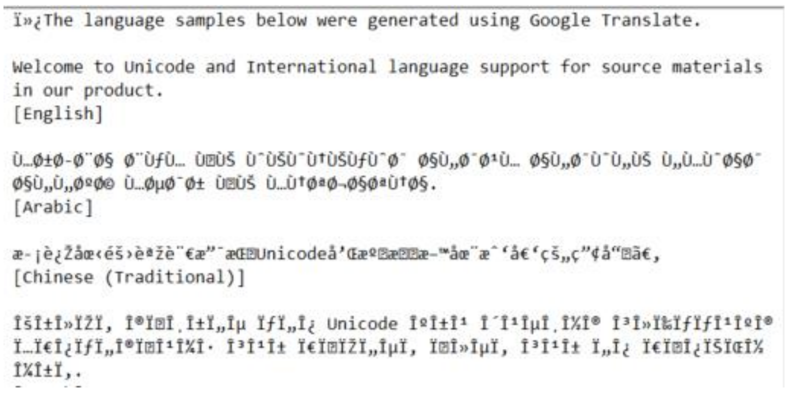
$2^1 = 2$	$2^9 = 512$
$2^2 = 4$	$2^{10} = 1024$
$2^3 = 8$	$2^{11} = 2048$
$2^4 = 16$	$2^{12} = 4096$
$2^5 = 32$	$2^{13} = 8192$
$2^6 = 64$	$2^{14} = 16384$
$2^7 = 128$	$2^{15} = 32768$
$2^8 = 256$	$2^{16} = 65536$

Numbering System Example

- What are the hexadecimal and binary equivalent of each of the following decimal numbers

DECIMAL	BINARY	HEXADECIMAL
4097	0001 0000 0000 0001	1001
27	0000 0000 0001 1011	001B
284	0000 0001 0001 1100	011C

Character Encoding – a Problem we have ALL experienced! Right?



```
ï»¿The language samples below were generated using Google Translate.

Welcome to Unicode and International language support for source materials
in our product.
[English]

Ù…Ø±ØØ¨Ø§ Ø¨ÙƒÙ… ÙÙŠ ÙˆÙŠÙˆÙ†ÙŠÙƒÙˆØ¯ Ø§Ù„Ø¯Ø¹Ù… Ø§Ù„Ø¯ÙˆÙ„ÙŠ Ù„Ù„Ù…ÙˆØ§Ø¯
Ø§Ù„Ù„ØºØ© Ù…ØµØ¯Ø± ÙÙŠ Ù…Ù†ØªØ¬Ø§ØªÙ†Ø§.
[Arabic]

æ¡è¿Žåœ¨é‡‹è¢«èˆ‡æ”¯æŒUnicodeå’Œæº▯æ▯▯æ–™åœ¨æˆ‘å€‘çš„ç”¢å“▯ã€‚
[Chinese (Traditional)]

ÎšÎ±Î»ÏŽÏ‚ Î®ÏÎ¸Î±Ï„Îµ ÏƒÏ„Î¿ Unicode ÎºÎ±Î¹ Î´Î¹ÎµÎ¸Î½Î® Î³Î»Ï‰ÏƒÏƒÎ¹ÎºÎ®
Ï…Ï€Î¿ÏƒÏ„Î®ÏÎ¹Î¾Î· Î³Î¹Î± Ï€ÏÏŽÏ„Î·Ï‚ ÏÎ»ÎµÏ‚ Î³Î¹Î± Ï„Î¿ Ï€ÏÎ¿ÏŠÏŒÎ½
Î½Î±Ï‚.
```

Character Encoding - Ideas and data are different

- One idea has many possible encodings
- An encoding is just a method to transform an idea (like the letter "A") into raw data (bits and bytes)
- The "idea" of "A" can be encoded many different ways
  - Encodings differ in efficiency and compatibility
- Embrace the philosophy that an <u>idea or concept and the data that store it are different</u>. Let that swirl around in your mind for a minute or two…

Character Encoding (CE) - Explained

- First, let me say while the CE problem is common it is NOT that hard to get right… So let's, shall we…
- What are characters? (Not a rhetorical question)
  - Bytes are bytes and characters are an abstraction…
- The idea of a character is an **agreement** (unenforceable!) that a certain number should be interpreted as a certain letter
  - That's it! That's the way in was in 1940 and the way it is today.
  - This agreement is a called an ***character map***
  - But today we all agree (mostly) on the mapping of numbers to characters. https://home.unicode.org/

Unicode

- 20+ year old non-profit (and benevolent overload) of all known characters
- In fact, Unicode has a different way of thinking about characters, and you have to understand the Unicode way of thinking of things or nothing will make sense
- Until now, we've assumed that a letter maps to some bits which you can store on disk or in memory. Example:

```
A -> 0x41 -> 0b01000001
```

Unicode

- In Unicode, a letter maps to something called a *code point* which is still just a theoretical concept
- In Unicode, the letter *A* is a platonic ideal. It's just floating in heaven
- This platonic A is different than B, and different from *a*, but the same as A and ***A*** and A (A in 3 different fonts)
  - Think of fonts as "formatting". 1 = $1 = $1.00, etc. These are all equal to 1, they just look different
- The actual on-screen representation of code points are called glyphs
- The **complete mapping of code points to glyphs is known as a font**

$$\begin{tikzcd} \boxed{\text{Code Point}} \arrow[r, "\text{Font}"] & \boxed{\text{Glyph}} \end{tikzcd}$$

Unicode

- Every platonic letter in every alphabet is assigned a magic number by the Unicode consortium known as a "Code Point"
- Code Points are written like this: **U+0048**
  - U+ means "Unicode" and the <u>numbers are hexadecimal</u>
    - See, I told you knowing numbering systems was going to be useful
  - Type *charmap* Windows Taskbar search box. Let's explore this together for second…
  - Examine the "exclamation mark" across all fonts – what do you make of what you see?
    - Keep your eye on the U+0021 value and what the glyph looks like across fonts
    - Which fonts produce a glyph for U+0021 that looks radically different from the rest?

Unicode

- OK, so say we have a string: `‘Hello’`
  - which, in Unicode, corresponds to these five code points:
  - U+0048 U+0065 U+006C U+006C U+006F
  - Just a bunch of code points (numbers)
  - We haven’t yet said anything about how to store this in memory or represent it in an email message
- Enter, *encodings*…

Encodings

- Ok, got it, code points are an abstraction
  - Remember, bytes are bytes and characters are an abstraction
- code points require a **character encoding** to convert them into the <u>one thing which all computers can interpret – bytes</u>
- Once converted to bytes, code points can be saved to files or sent over the network

Encodings

- **Encodings are just an agreement** about how character data should be interpreted as it is being *disgorged* (written) or *ingested (read)*
- The bottom line – <u>if you don't know how a file was originally encoded it can be very difficult (if not impossible) to accurately read it's contents</u>

Encoding

- There are **hundreds** of historical (non-Unicode) encodings which can only store *some* code points correctly and change all the other code points into question-mark laden gibberish
- **Example**: Popular encoding of English text are Windows-1252 (the Windows 9x standard) and ISO-8859-1 (aka Latin-1)
  - FYI, Windows-1252 is a superset of ISO-8859-1
- Try to store Russian or Hebrew code points using these encodings and you get a bunch of question marks.

UTF-8 Encoding

- UTF-8 (UNICODE Transformation Format-8) is currently the world's most popular character encoding
- UTF-8 uses a set of rules to convert a code point into an unique sequence of (1 to 4) bytes, and vice versa
- Code points are said to be **encoded** into a sequence of bytes
  - I.e., when writing Unicode data
- Sequences of bytes are **decoded** into code points
  - I.e., when reading Unicode data

Encodings – finally a bit of luck

- Open the Chrome or Firefox browser to www.duq.edu and press ***ctrl+u***
  - See anything that might be useful in helping the browser determine encoding of the HTML text of the web page?

```
<meta charset="utf-8" />
```

- UTF-8 is (by far) the most popular encoding in use today
- Most important text based file formats (HTML, XML, JSON, RDF, RDF-LD) include information about their encoding
- BTW, anyone see a chicken-and-egg problem here?

Little and Big Endian?

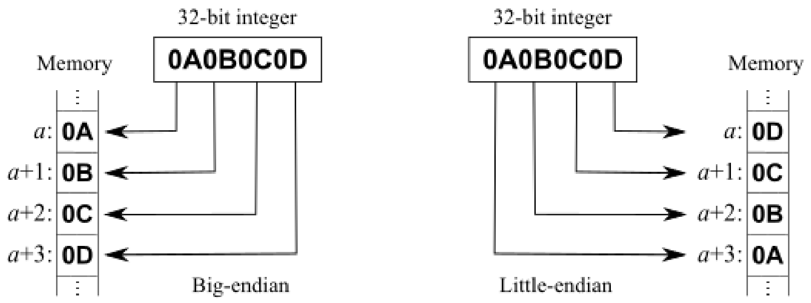
What happened to our luck?

Little and Big Endian Mystery

- Little and big endian are two ways of storing multi-byte data-types
  - A function of the CPU hardware of your machine
- In little endian machines, the least significant byte (lowest power position) of binary representation of the **multibyte** data-type is stored first
- In big endian machines, the most significant byte (highest power position) of binary representation of the multibyte data-type is stored first
- Can you see how this would impact encoding? Good… How exactly?

Little and Big Endian Example

- Suppose an integer number is stored as a multi-byte data type. A variable x with value 0x0A0B0C0D will be stored as following.



Endianness Impacts on Reading Unicode

- Since <u>bytes are bytes and characters are an abstraction</u>, in order to correctly interpret multi-byte UNICODE character data we MUST know the endianness that was used to encode the file
- If we don't, **we will not be able to interpret the data correctly as characters**

Enter Byte Order Marks (BOM)…

- Byte Order Marks (BOMs) can (and should) be included at the beginning of a UNICODE file to indicate endianness
- Open https://hexed.it/?hl=en in your browser
- Download all of the files from the *Character Encoding / Encoding Lab* item
- Use *hexed.it* to open the UTF8_BOM.txt file

EF BB EF is the (optional) BOM for UTF8

- FYI, EF BB EF, does not indicate endianness as UTF8 is byte "centric"
  - Being byte centric is part of the UTF8 encoding standard
  - When you output UTF8 you output only bytes – **never** multi-byte data types
- Now open each of the UTF-16*.txt files.
- What do you see?
- Using the file names as a guide, what is the BOM for files stored as
  - Big Endian? (aka "*network byte order*")
  - Little Endian?

The Takeaway…

- You **must** communicate to Python a file's encoding
  - Not surprisingly, the default is UTF-8
- It is (normally) NOT necessary to communicate the "endianness" of a file as BOMs are well defined and the Python IO system knows how to deal with them
  - This is a nice feature of Python and may or may not be true of any other text manipulation tools and/or programming language environments
- **NOTABLE EXCEPTION**: If a BOM is included in a UTF-8 encoded file(I'm looking at you Bill Gates…) you must communicate this to Python
  - Why? Because UTF-8 files are not expected to have BOMs since they are byte centric and cannot manifest an endianness problem. Historically, Microsoft programs have been the biggest offender when it comes to this behavior.

Summary

- Character data is SUPREMELY important when wrangling data from most all data sources
  - The reason for this will become clear as you work through next weeks activities
  - By extension, this makes character encoding supremely important
- Understanding the role of UNICODE and character encoding in the “consumption” of that data is something you MUST get right

Try It.

- Open Anaconda Navigator from the Windows App menu
- Navigate to the *EncodingTest.ipynb* file previously downloaded
- Review the contents
- Run all cells…