

# File Formats

INF 551

Wensheng Wu

# File Formats

- Specify what information bits in file encode
- Example: text file
  - String of characters with particular **encoding** scheme, e.g., ASCII and Unicode
  - E.g., TXT, HTML, JSON, XML
- Others: xls, ppt, pdf, jpg, gif, mp3, png, etc.

# Roadmap

- Character encoding



- ASCII
- Unicode

- JSON

- XML (next lecture)

# Code space & points

- Code space
  - A range of numerical values available for encoding characters
  - E.g., 0 to 10FFFF for Unicode, 0 to 7F for ASCII
- Code point
  - A value for a character in a code space
- Unicode code point
  - U+ followed by its hexadecimal value, e.g., U+0058 for capital letter 'X')

# Encoding (of code points)

- Code unit: the smallest unit (comprising a number of bits) used to construct an encoding for a code point
  - Code unit for UTF-8: 8-bit
  - UTF-16: 16-bit
- UTF (Unicode Transformation Format) encoding
  - E.g., UTF-8 and UTF-16

# Variable-length encoding

- Different characters may be encoded using codes of different length
- In Unicode, a code point may be represented using multiple code units
  - E.g., 1-4 in UTF-8, 1-2 in UTF-16

# ASCII

- American Standard Code for Information Interchange
- 128 characters: 7-bit code (code points: 0~7F)
  - Digits: 0-9 (0x30 – 0x39)
  - Lowercase letters: a-z (0x61 – 0x7A)
  - Uppercase letters: A-Z (0x41 – 0x5A)
  - White space (0x20)
  - Punctuation symbols
  - Control characters (e.g., Ctrl-C: 0x03)

# ASCII

Dec	Hex	Name	Char	Ctrl-char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
0	0	Null	NUL	CTRL-@	32	20	Space	64	40	@	96	60	`
1	1	Start of heading	SOH	CTRL-A	33	21	!	65	41	A	97	61	a
2	2	Start of text	STX	CTRL-B	34	22	"	66	42	B	98	62	b
3	3	End of text	ETX	CTRL-C	35	23	#	67	43	C	99	63	c
4	4	End of xmit	EOT	CTRL-D	36	24	\$	68	44	D	100	64	d
5	5	Enquiry	ENQ	CTRL-E	37	25	%	69	45	E	101	65	e
6	6	Acknowledge	ACK	CTRL-F	38	26	&	70	46	F	102	66	f
7	7	Bell	BEL	CTRL-G	39	27	'	71	47	G	103	67	g
8	8	Backspace	BS	CTRL-H	40	28	(	72	48	H	104	68	h
9	9	Horizontal tab	HT	CTRL-I	41	29	)	73	49	I	105	69	i
10	0A	Line feed	LF	CTRL-J	42	2A	*	74	4A	J	106	6A	j
11	0B	Vertical tab	VT	CTRL-K	43	2B	+	75	4B	K	107	6B	k
12	0C	Form feed	FF	CTRL-L	44	2C	,	76	4C	L	108	6C	l
13	0D	Carriage feed	CR	CTRL-M	45	2D	-	77	4D	M	109	6D	m
14	0E	Shift out	SO	CTRL-N	46	2E	.	78	4E	N	110	6E	n
15	0F	Shift in	SI	CTRL-O	47	2F	/	79	4F	O	111	6F	o
16	10	Data line escape	DLE	CTRL-P	48	30	0	80	50	P	112	70	p
17	11	Device control 1	DC1	CTRL-Q	49	31	1	81	51	Q	113	71	q
18	12	Device control 2	DC2	CTRL-R	50	32	2	82	52	R	114	72	r
19	13	Device control 3	DC3	CTRL-S	51	33	3	83	53	S	115	73	s
20	14	Device control 4	DC4	CTRL-T	52	34	4	84	54	T	116	74	t
21	15	Neg acknowledge	NAK	CTRL-U	53	35	5	85	55	U	117	75	u
22	16	Synchronous idle	SYN	CTRL-V	54	36	6	86	56	V	118	76	v
23	17	End of xmit block	ETB	CTRL-W	55	37	7	87	57	W	119	77	w
24	18	Cancel	CAN	CTRL-X	56	38	8	88	58	X	120	78	x
25	19	End of medium	EM	CTRL-Y	57	39	9	89	59	Y	121	79	y
26	1A	Substitute	SUB	CTRL-Z	58	3A	:	90	5A	Z	122	7A	z
27	1B	Escape	ESC	CTRL-[	59	3B	;	91	5B	[	123	7B	{
28	1C	File separator	FS	CTRL-\	60	3C	<	92	5C	\	124	7C	
29	1D	Group separator	GS	CTRL-]	61	3D	=	93	5D	]	125	7D	}
30	1E	Record separator	RS	CTRL-^	62	3E	>	94	5E	^	126	7E	~
31	1F	Unit separator	US	CTRL-`	63	3F	?	95	5F	_	127	7F	DEL



# Windows-1253

- Windows code page for Latin + Greek characters
- Use 8 bits
  - 0x00 ~ 0xFF

Codepage 1253 - Greece Windows

	-0	-1	-2	-3	-4	-5	-6	-7	-8	-9	-A	-B	-C	-D	-E	-F
0-		0001	0002	0003	0004	0005	0006	0007	0008	0009	000A	000B	000C	000D	000E	000F
1-	0010	0011	0012	0013	0014	0015	0016	0017	0018	0019	001A	001B	001C	001D	001E	001F
2-		!	"	#	\$	%	&	'	(	)	*	+	,	-	.	/
3-	0020	0021	0022	0023	0024	0025	0026	0027	0028	0029	002A	002B	002C	002D	002E	002F
4-	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
5-	0030	0031	0032	0033	0034	0035	0036	0037	0038	0039	003A	003B	003C	003D	003E	003F
6-	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
7-	0040	0041	0042	0043	0044	0045	0046	0047	0048	0049	004A	004B	004C	004D	004E	004F
8-	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
9-	0050	0051	0052	0053	0054	0055	0056	0057	0058	0059	005A	005B	005C	005D	005E	005F
A-	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
B-	0060	0061	0062	0063	0064	0065	0066	0067	0068	0069	006A	006B	006C	006D	006E	006F
C-	p	q	r	s	t	u	v	w	x	y	z	{		}	~	
D-	0070	0071	0072	0073	0074	0075	0076	0077	0078	0079	007A	007B	007C	007D	007E	007F
E-	€	,	f	”	...	†	‡		‰	<						
F-	20AC	0081	0192	201A	201E	2026	2020	2021	0088	2030	008A	2039	008C	008D	008E	008F
G-	‘	’	“	”	•	—			™			>				
H-	0090	2018	2019	201C	201D	2022	2013	2014	0098	2122	009A	203A	009C	009D	009E	009F
I-	’	À	£	¤	¥	¦	§	¨	©	ª	«	¬	®	¯		
J-	00A0	0385	0386	00A3	00A4	00A5	00A6	00A7	00A8	00A9	00AA	00AB	00AC	00AD	00AE	2015
K-	°	±	²	³	´	µ	¶	·	¸	¹	º	»	¼	½	¾	¸
L-	00B0	00B1	00B2	00B3	00B4	00B5	00B6	00B7	0388	0389	038A	00BB	038C	00BD	038E	038F
M-	ı	Α	Β	Γ	Δ	Ε	Ζ	Η	Θ	Ι	Κ	Λ	Μ	Ν	Ξ	Ο
N-	0390	0391	0392	0393	0394	0395	0396	0397	0398	0399	039A	039B	039C	039D	039E	039F
O-	Π	Ρ		Σ	Τ	Υ	Φ	Χ	Ψ	Ω	İ	ÿ	ά	έ	ή	ί
P-	03A0	03A1		03A3	03A4	03A5	03A6	03A7	03A8	03A9	03AA	03AB	03AC	03AD	03AE	03AF
Q-	ϐ	α	β	γ	δ	ε	ζ	η	θ	ι	κ	λ	μ	ν	ξ	ο
R-	03B0	03B1	03B2	03B3	03B4	03B5	03B6	03B7	03B8	03B9	03BA	03BB	03BC	03BD	03BE	03BF
S-	π	ρ	ς	σ	τ	υ	φ	χ	ψ	ω	ϊ	ϋ	ό	ύ	ώ	
T-	03C0	03C1	03C2	03C3	03C4	03C5	03C6	03C7	03C8	03C9	03CA	03CB	03CC	03CD	03CE	

# Unicode

- Unicode supports more characters than ASCII and various code pages
- Unicode separates code points from encoding
  - In contrast to ASCII, where code point = encoding

# Unicode

- Code space is divided into 17 planes
- Each plane = contiguous  $2^{16}$  code points
- Recall that code points range from 0 to 10FFFF

⇒ Total code points =  $17 * 2^{16}$  or 1,114,112  
code points

Note  $2^{16} = 65,536$

# Planes in Unicode

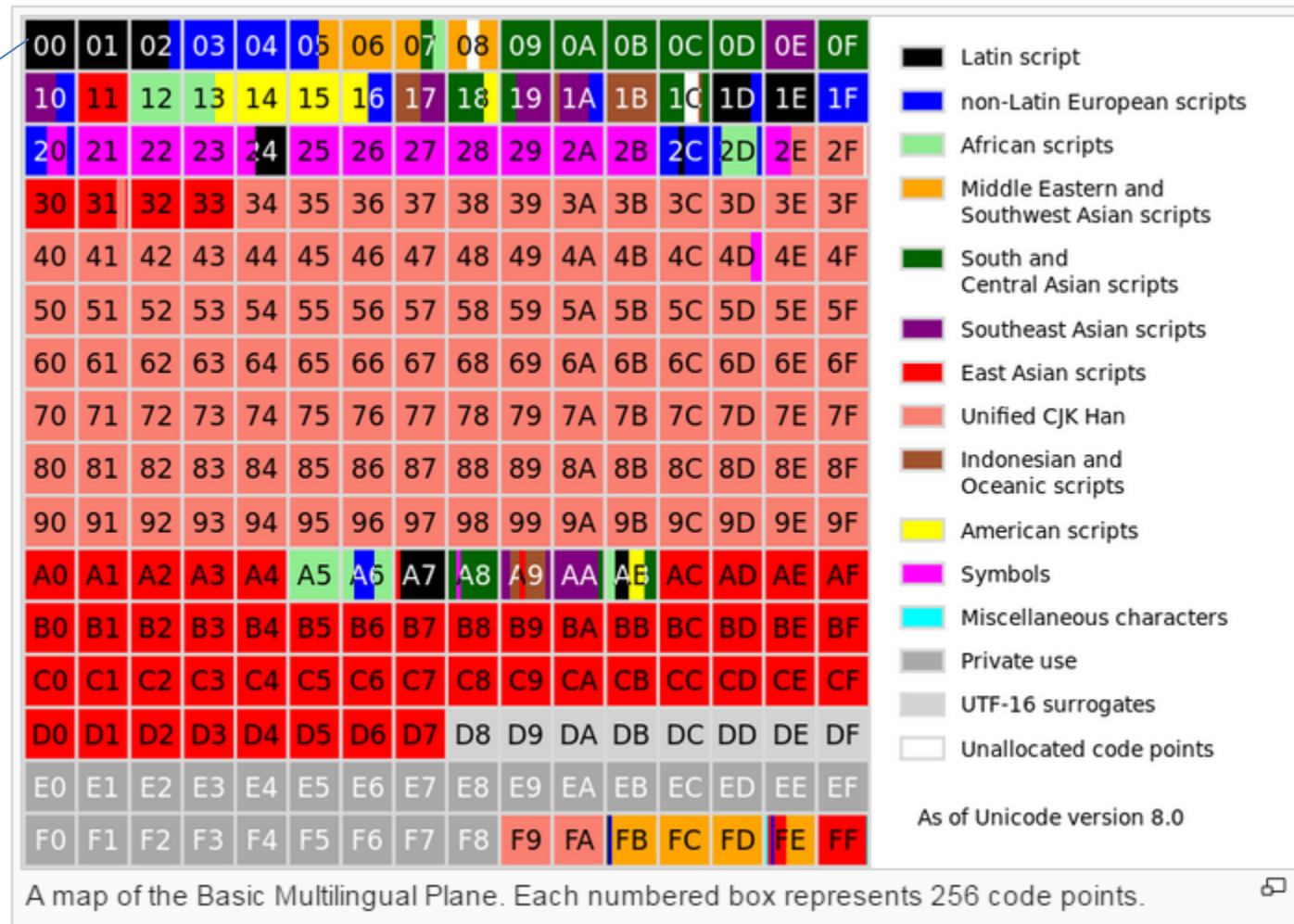
V·T·E

## Unicode planes and used code point ranges

[hide]

Basic		Supplementary						
Plane 0		Plane 1		Plane 2		Planes 3–13	Plane 14	Planes 15–16
0000–FFFF		10000–1FFFF		20000–2FFFF		30000–DFFFF	E0000–EFFFF	F0000–10FFFF
Basic Multilingual Plane		Supplementary Multilingual Plane		Supplementary Ideographic Plane		unassigned	Supplement-ary Special-purpose Plane	Supplement-ary Private Use Area
BMP		SMP		SIP		—	SSP	S PUA A/B
0000–0FFF	8000–8FFF	10000–10FFF		20000–20FFF	28000–28FFF		E0000–E0FFF	15: PUA-A
1000–1FFF	9000–9FFF	11000–11FFF		21000–21FFF	29000–29FFF		F0000–FFFFF	
2000–2FFF	A000–AFFF	12000–12FFF		22000–22FFF	2A000–2AFFF			
3000–3FFF	B000–BFFF	13000–13FFF	1B000–1BFFF	23000–23FFF	2B000–2BFFF			16: PUA-B
4000–4FFF	C000–CFFF	14000–14FFF		24000–24FFF	2C000–2CFFF			100000–10FFFF
5000–5FFF	D000–DFFF		1D000–1DFFF	25000–25FFF				
6000–6FFF	E000–EFFF	16000–16FFF	1E000–1EFFF	26000–26FFF				
7000–7FFF	F000–FFFF		1F000–1FFFF	27000–27FFF	2F000–2FFFF			

# Plane 0: BMP (Basic Multilingual Plane)



Each block represents 256 code points

# UTF-8

- Encoding scheme for Unicode code space
- Code unit = 8 bits
- Variable length
  - Code point may be represented using 1-4 code units

# UTF-8 Design

- This shows the original design
  - Current: only up to U+10FFFF code points used
  - So no 5-byte/6-byte sequences

Bits of code point	First code point	Last code point	Bytes in sequence	Byte 1	Byte 2	Byte 3	Byte 4	Byte 5	Byte 6
7	U+0000	U+007F	1	0xxxxxxx					
11	U+0080	U+07FF	2	110xxxxx	10xxxxxx				
16	U+0800	U+FFFF	3	1110xxxx	10xxxxxx	10xxxxxx			
21	U+10000	U+1FFFFF	4	11110xxx	10xxxxxx	10xxxxxx	10xxxxxx		
26	U+200000	U+3FFFFFFF	5	111110xx	10xxxxxx	10xxxxxx	10xxxxxx	10xxxxxx	
31	U+4000000	U+7FFFFFFF	6	1111110x	10xxxxxx	10xxxxxx	10xxxxxx	10xxxxxx	10xxxxxx

# UTF-8 Features

- Backward compatibility
  - One byte for ASCII, leading bit of byte is zero
- Clear distinction btw single- vs. multi-byte characters
  - Single-byte/multi-byte: start with 0/1 respectively
- Multiple length
  - a leading byte starts with 2 or more 1's, followed by a 0, e.g., '110', '1110', etc.
  - One or more continuation bytes all start with '10'



# UTF-8 Features

- Clear indication of code sequence length
  - By # of 1's in leading byte (for multi-byte)
- Self-synchronization
  - Can find start of characters by backing up at most 3 bytes (5 in original design)

# Example

- Encode '€' using UTF-8
- Code point = U+20AC
- Need 3 bytes in UTF-8

Character		Binary code point	Binary UTF-8	Hexadecimal UTF-8
\$	U+0024	0100100	00100100	24
¢	U+00A2	00010100010	11000010 10100010	C2 A2
€	U+20AC	0010000010101100	11100010 10000010 10101100	E2 82 AC
☺	U+10348	000010000001101001000	11110000 10010000 10001101 10001000	F0 90 8D 88

# UTF-16

- Code unit = 16 bits
- Variable-length encoding
  - Code point = one/two code units
- Not compatible with ASCII

# UTF-16

- Plane 0: encoded using one code unit: 16 bit
- Rest: two code units

Unicode planes and used code point ranges <span>[hide]</span>									
Basic		Supplementary							
Plane 0		Plane 1		Plane 2		Planes 3–13	Plane 14	Planes 15–16	
0000–FFFF		10000–1FFFF		20000–2FFFF		30000–DFFFF	E0000–EFFFF	F0000–10FFFF	
Basic Multilingual Plane		Supplementary Multilingual Plane		Supplementary Ideographic Plane		<i>unassigned</i>	Supplement-ary Special-purpose Plane	Supplement-ary Private Use Area	
BMP		SMP		SIP		—	SSP	S PUA A/B	
0000–0FFF	8000–8FFF	10000–10FFF		20000–20FFF	28000–28FFF		E0000–E0FFF	15: PUA-A F0000–FFFFF	
1000–1FFF	9000–9FFF	11000–11FFF		21000–21FFF	29000–29FFF				
2000–2FFF	A000–AFFF	12000–12FFF		22000–22FFF	2A000–2AFFF				
3000–3FFF	B000–BFFF	13000–13FFF	1B000–1BFFF	23000–23FFF	2B000–2BFFF			16: PUA-B 100000– 10FFFFF	
4000–4FFF	C000–CFFF	14000–14FFF		24000–24FFF	2C000–2CFFF				
5000–5FFF	D000–DFFF		1D000–1DFFF	25000–25FFF					
6000–6FFF	E000–EFFF	16000–16FFF	1E000–1EFFF	26000–26FFF					
7000–7FFF	F000–FFFF		1F000–1FFFF	27000–27FFF	2F000–2FFFF				

# UTF-16 Encoding

- U+0000 to U+D7FF and U+E000 to U+FFFF
  - One code unit, i.e., 2 bytes
- U+D800 to U+DFFF
  - Reserved
- U+10000 to U+10FFFF
  - Two code units, i.e., 4 bytes

# Encoding planes 1 to 16

- Code points: 10000 to 10FFFF
1. Subtract 10000 from code point  
=> 0..FFFFF (20 bits)
  2. Add 1<sup>st</sup> 10 bits to D800 => 1<sup>st</sup> code unit
  3. Add 2<sup>nd</sup> 10 bits to DC00 => 2<sup>nd</sup> code unit

# Examples

- Encoding code points in BMP is easy

Character	Binary code point	Binary UTF-16	UTF-16 hex code units	UTF-16BE hex bytes	UTF-16LE hex bytes
\$ U+0024	0000 0000 0010 0100	0000 0000 0010 0100	0024	00 24	24 00
€ U+20AC	0010 0000 1010 1100	0010 0000 1010 1100	20AC	20 AC	AC 20
¥ U+10437	0001 0000 0100 0011 0111	1101 1000 0000 0001 1101 1100 0011 0111	D801 DC37	D8 01 DC 37	01 D8 37 DC
𐤆 U+24B62	0010 0100 1011 0110 0010	1101 1000 0101 0010 1101 1111 0110 0010	D852 DF62	D8 52 DF 62	52 D8 62 DF

Subtract 0001 from 0010 => 0001

# Decoding UTF-16 (big endian)

- Divide file into 2-byte words
- If the word starts with 1101 10xx
  - Must be the first code unit of a 2-unit encoding
- Else if the word starts with 1101 11xx
  - Must be the second code unit of a 2-unit encoding
- Else
  - Must be a 1-unit encoding



# Big-Endian (BE) and Little-Endian (LE)

- Two ways of storing a multi-byte word (code unit)
  - Not a problem in UTF-8
- UTF-16BE
  - BE: big endian where most significant bytes stored first
  - So no change to the order of bytes
  - So ABCD stored as “AB CD”
- UTF-16LE
  - Reverse the order
  - ABCD stored as “CD AB”

# Byte Order Mark (BOM)

- Unicode recommends to add BOM to the beginning of a text
  - Tell which order the text follows
  - BOM: U+FEFF
- “FE FF” => big endian
- “FF FE” => small endian

# Gulliver's Travels

Besides, our Histories of six thousand Moons make no mention of any other Regions, than the two great Empires of Lilliput and Blefuscu. Which two mighty Powers have, as I was going to tell you, been engaged in a most obstinate War for six and thirty Moons past.

It began upon the following Occasion. It is allowed on all Hands, that the primitive way of breaking Eggs, before we eat them, was upon the larger End: But his present Majesty's Grand-father, while he was a Boy, going to eat an Egg, and breaking it according to the ancient Practice, happened to cut one of his Fingers. Whereupon the Emperor his Father published an Edict, commanding all his Subjects, upon great Penaltys, to break the smaller End of their Eggs.

The People so highly resented this Law, that our Histories tell us there have been six Rebellions raised on that account; wherein one Emperor lost his Life, and another his Crown. These civil Commotions were constantly fomented by the Monarchs of Blefuscu; and when they were quelled, the Exiles always fled for Refuge to that Empire. It is computed, that eleven thousand Persons have, at several times, suffered Death, rather than submit to break their Eggs at the smaller End.

Many hundred large Volumes have been published upon this Controversy: But the books of the **Big-Endians** have been long forbidden, and the whole Party rendered incapable by Law of holding Employments. During the Course of these Troubles, the Emperors of Blefuscu did frequently expostulate by their Ambassadors, accusing us of making a Schism in Religion, by offending against a fundamental Doctrine of our great Prophet Lustrog, in the fifty-fourth Chapter of the Brundrecal (which is their Alcoran.) This, however, is thought to be a meer Strain upon the Text: For the Words are these: That all true Believers shall break their Eggs at the convenient End: and which is the convenient End, seems, in my humble Opinion, to be left to every Man's Conscience, or at least in the power of the Chief Magistrate to determine.

Now the Big-Indian Exiles have found so much Credit in the Emperor of Blefuscu's Court, and so much private Assistance and Encouragement from their Party here at home, that a bloody War has been carried on between the two Empires for six and thirty Moons with various Success; during which time we have lost forty Capital Ships, and a much greater number of smaller Vessels, together with thirty thousand of our best Seamen and Soldiers; and the Damage received by the Enemy is reckon'd to be somewhat greater than Ours. However, they have now equipped a numerous Fleet, and are just preparing to make a Descent upon us; and his Imperial Majesty, placing great Confidence in your Valour and Strength, has commanded me to lay this Account of his affairs before you.

# "Example" in different encodings

ASCII: 45 78 61 6d 70 6c 65

UTF-16BE: FE FF 00 45 00 78 00 61 00 6d 00 70 00 6c 00 65

UTF-16LE: FF FE 45 00 78 00 61 00 6d 00 70 00 6c 00 65 00

# What about end of line?

- LF (Line feed, '\n', 0x0A, 10 decimal)
- CR (Carriage return, '\r', 0x0D, 13 in decimal)
- Different systems represent it differently
  - Window: \r\n
  - Unix based: \n
  - Old Mac: \r

# A type writer

Carriage return lever

ribbon

Lever

Paper

Carriage

Type hammer




# Unicode

Unicode standard defines a number of characters that conforming applications should recognize as line terminators:

- LF: Line Feed, U+000A
- VT: Vertical Tab, U+000B
- FF: Form Feed, U+000C
- CR: Carriage Return, U+000D
- CR+LF: CR (U+000D) followed by LF (U+000A)
- NEL: Next Line, U+0085
- LS: Line Separator, U+2028
- PS: Paragraph Separator, U+2029

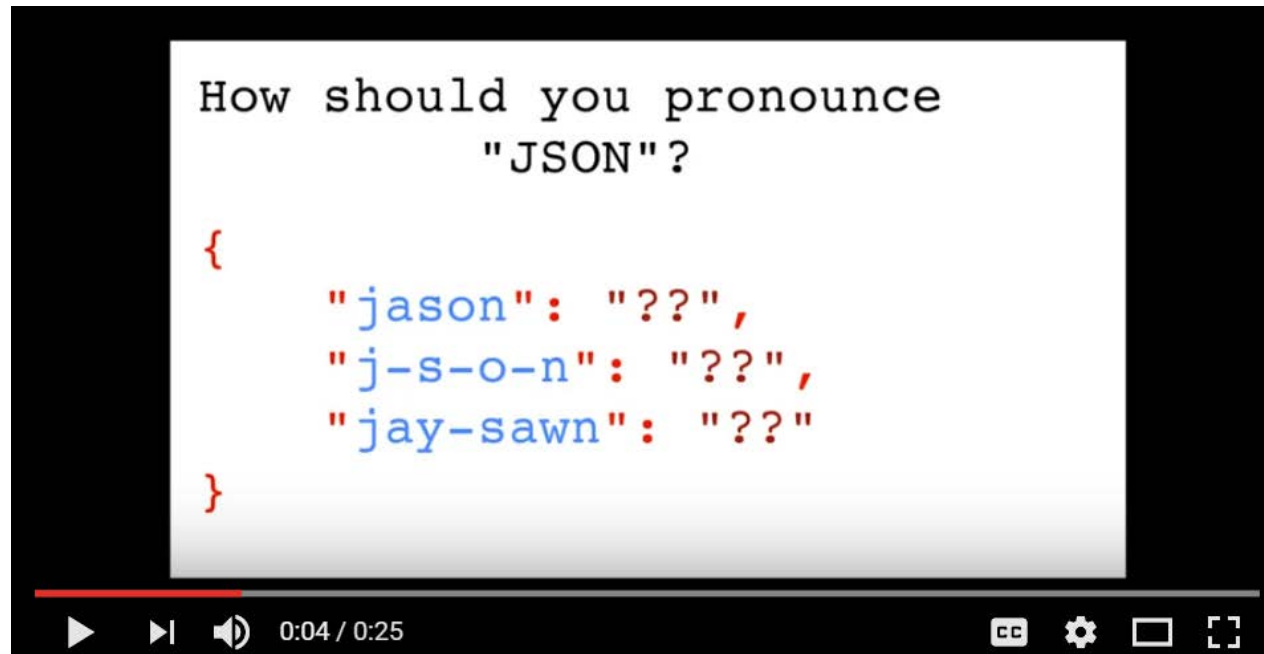
# Roadmap

- Character encoding
  - ASCII
  - Unicode
- JSON 
- XML



# How to pronounce JSON??

- Douglas Crockford: pronouncing "JSON"
  - <https://www.youtube.com/watch?v=zhVdWQWKRqM>



# JSON (Javascript Object Notation)

- Light-weight data exchange format
  - Much simpler than XML
  - Language-independent
  - Inspired by syntax of JavaScript object literals
- Some differences from JavaScript objects, e.g.,
  - String in JSON must be double-quoted
  - Ok to single-quote in JavaScript (& Python)

# Syntax of JSON

- value =  
string | number | object | array | true | false | null
- object = { } | { members }
  - members = pair | pair, members
  - pair = string : value
- array = [ ] | [ elements ]
  - elements = value | value, elements

# Valid JSON or not?

- []
- {}
- {[]}
- [{}]
- {"name": john}
- {name: "john"}
- {"name": 25}
- "name"
- 25
- {25}
- [25]


# JSON is case-sensitive

- Valid or not?
  - True
  - true
  - TRUE
  - Null
  - false


# Example JSON

```
{
  "firstName": "John",
  "lastName": "Smith",
  "isAlive": true,
  "age": 25,
  "address": {
    "streetAddress": "21 2nd Street",
    "city": "New York",
    "state": "NY",
    "postalCode": "10021-3100"
  },
  "phoneNumbers": [
    {
      "type": "home",
      "number": "212 555-1234"
    },
    {
      "type": "office",
      "number": "646 555-4567"
    }
  ],
  "children": [],
  "spouse": null
}
```

Value is an object



Value is an array



# Check syntax of JSON

- JSON validator
  - <http://jsonlint.com/>

# Python primer

- String: 'abc' or "abc"
- List: `x = ['abc', 25]`
  - `x[0] = ?`
  - `x.append(True)` // True is boolean
  - `x.append(None)` // None is `NoneType` = null
- Tuple: `y = ('abc', 25)`
  - `y[0] = ?`
  - What about `y.append(True)`?



# Python primer

- Dictionary: `z = {'name': 'john', 25: 'age'}`
  - Note key in Python can also be integer or tuple
  - `z['name'] = ?`
  - `z[25]=?`
  - What about `z['age']` or `z['25']`?
- `z['gender'] = 'male'`
  - `z = {25: 'age', 'name': 'john', 'gender': 'male'}`

# Working with JSON in Python

- `import json`
  - Loading json library module
- `json.dumps()`
  - JSON encoder
  - Python object => JSON value
- `json.loads()`
  - JSON decoder
  - JSON value => Python object

# Python object => JSON value

- Python list => JSON array
  - `json.dumps([1, 2]) => '[1, 2]'`
  - `json.dumps([3, 'abc', True, None]) => '[3, "abc", true, null]'`
- Python tuple => JSON array
  - `json.dumps((1, 'abc')) => '[1, "abc"]'`

# Python object => JSON value

- Python dictionary => JSON object
  - `json.dumps({'name': 'john', 25: 'age'}) => '{"25": "age", "name": "john"}'`
- Notes
  - `None` => `null`
  - `True` => `true`
  - `'abc'` => `"abc"`

# Python object => JSON value

- `json.dumps(['foo', {'bar': ('baz', None, 1.0, 2)}])`
  - `'["foo", {"bar": ["baz", null, 1.0, 2]}]'`
- `json.dumps({(1,2): 5})`
  - Error (key is a tuple, Ok in Python)
  - `dumps()` doesn't take tuple as key (but see below)
- `json.dumps({(2): 5}) => '{"2": 5}'`

# JSON value => Python

- JSON object => Python dictionary
  - `json.loads('{"name": "john", "age": 5}')` => `{u'age': 5, u'name': u'john'}`
  - Note: 'u' means "Unicode"
- JSON array => Python list
  - `json.loads('[25, "abc"]')` => `[25, u'abc']`

# JSON value => Python

- `json.loads('\"abc\"')` => `u'abc'`
- `json.loads('25.2')` => `25.2`
- `json.loads('true')` => `True`
- `json.loads('null')` => `None`
  
- `json.loads('{\"name\": \"john\", \"age\": 25, \"phone\": [123, 456]})'`  
=> `{u'phone': [123, 456], u'age': 25, u'name': u'john'}`

# Conversion summary

JSON	Python
Object	Dictionary
Array	List
Array	Tuple
null	None
true	True
false	False

Python dictionary => JSON object

- Keys in Python can be number, string, or tuple.
- Number is also converted to string.
- But tuple (with two or more components) is not acceptable by dumps()/dump().



# Working with files

- `f = open('lax.json')`
- `lax = json.load(f)`
- `out_file = open('output.json', 'w')`
- `json.dump(lax, out_file)`

# LAX passenger traffic data

- Download data at:
  - <https://data.lacity.org/A-Prosperous-City/Los-Angeles-International-Airport-Passenger-Traffi/g3qu-7q2u>
- A copy is available on Blackboard too
  - In the Resources folder

# Data spreadsheet

	ReportPeriod ⓘ ⋮	Terminal	Arrival_Departure ⓘ ⋮	Domestic_International	Passenger_Count ⓘ ⋮
4650 ⓘ	07/01/2016 12:00:00 AM	Terminal 3	Arrival	Domestic	402,452
4651 ⓘ	07/01/2016 12:00:00 AM	Terminal 3	Departure	Domestic	390,418
4652 ⓘ	07/01/2016 12:00:00 AM	Terminal 3	Departure	International	4,365
4653 ⓘ	07/01/2016 12:00:00 AM	Terminal 4	Arrival	Domestic	422,257
4654 ⓘ	07/01/2016 12:00:00 AM	Terminal 4	Arrival	International	35,800
4655 ⓘ	07/01/2016 12:00:00 AM	Terminal 4	Departure	Domestic	384,861
4656 ⓘ	07/01/2016 12:00:00 AM	Terminal 4	Departure	International	64,590
4657 ⓘ	07/01/2016 12:00:00 AM	Terminal 5	Arrival	Domestic	456,188
4658 ⓘ	07/01/2016 12:00:00 AM	Terminal 5	Arrival	International	78,781
4659 ⓘ	07/01/2016 12:00:00 AM	Terminal 5	Departure	Domestic	478,348
4660 ⓘ	07/01/2016 12:00:00 AM	Terminal 5	Departure	International	71,021
4661 ⓘ	07/01/2016 12:00:00 AM	Terminal 6	Arrival	Domestic	374,394

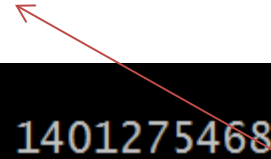
# JSON file (lax.json)

- Ignore "meta" info (in the beginning of file)
- Records are in the value of "data"

```
  },  
  "data" : [ [ 1, "31CAC749-9F88-4EA3-8EC0-0DFC3EE6DA81", 1, 1401275468, "883844",  
    "1401275468", "883844", "{\\n}", "2014-05-01T00:00:00", "2006-01-01T00:00:00", "Imperial Terminal", "Arrival", "Domestic", "490" ],  
    [ 2, "085E4C26-9CE8-41F1-AD72-7A8E42DCB3B9", 2, 1401275468, "883844", 1401275468, "883844", "{\\n}", "2014-05-01T00:00:00", "2006-01-01T00:00:00", "Imperial Terminal", "Departure", "Domestic", "498" ],  
    [ 3, "5EA27B8A-859C-4FA9-B697-38B7292E3689", 3, 1401275468, "883844", 1401275468, "883844", "{\\n}", "2014-05-01T00:00:00", "2006-01-01T00:00:00", "Misc. Terminal", "Arrival", "Domestic", "753" ],  
    [ 4, "7E96DA71-1DCF-4DF6-98E6-3BD3AF165821", 4, 1401275468, "883844", 1401275468, "883844", "{\\n}", "2014-05-01T00:00:00", "2006-01-01T00:00:00", "Misc. Terminal", "Departure", "Domestic", "688" ],  
    [ 5, "0FEAB567-5777-4FC4-9D0F-8C1B67CA44D7", 5, 1401275468, "883844", 1401275468, "883844", "{\\n}", "2014-05-01T00:00:00", "2006-01-01T00:00:00", "Terminal 1", "Arrival", "Domestic", "401535" ] ] ]
```

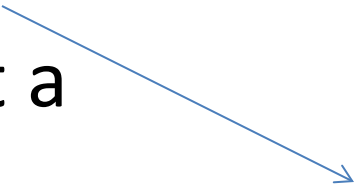
# Querying it in Python

ReportPeriod



```
>>> lax["data"][0]
[1, u'31CAC749-9F88-4EA3-8EC0-0DFC3EE6DA81', 1, 1401275468, u'883844', 1
401275468, u'883844', u'{\n}', u'2014-05-01T00:00:00', u'2006-01-01T00:0
0:00', u'Imperial Terminal', u'Arrival', u'Domestic', u'490']
>>> lax["data"][0][9]
u'2006-01-01T00:00:00'
>>> lax["data"][0][10]
u'Imperial Terminal'
>>> lax["data"][0][11]
u'Arrival'
>>> lax["data"][0][12]
u'Domestic'
>>> lax["data"][0][13]
u'490'
>>> |
```

# Unicode in Python

- `>>> a = u'\u20AC'` # note need u before '  
• `>>> print a`  
• €
- 

u indicates it is a Unicode string

# Resources

- UTF-8
  - <https://en.wikipedia.org/wiki/UTF-8>
- UTF-16
  - <https://en.wikipedia.org/wiki/UTF-16>
- JSON
  - <https://en.wikipedia.org/wiki/JSON>
  - Syntax: <http://www.json.org/>

# Resources

- JSON encoder and decoder
  - <https://docs.python.org/2/library/json.html>