Bits, bytes, and Unicode: An introduction to digital text for linguists

James A. Crippen[*]

26 September 2012

1. Writing

First I'll quickly and loosely define some properties of written language that are independent of digital text.

- a GRAPH is a unit of written linguistic communication
  - this is analogous to a 'phone', a unit of linguistic sound
  - examples: ⟨x⟩, ⟨X⟩, ⟨𝑥⟩, ⟨𝑋⟩, ⟨𝗑⟩, ⟨𝖷⟩, ⟨𝐱⟩, ⟨𝐗⟩, ⟨𝓍⟩, ⟨𝒳⟩, ⟨𝔵⟩, ⟨𝔛⟩, ⟨𝕩⟩, ⟨𝕏⟩, …
- a GRAPHEME is an abstract category representing a set of graphs that are related somehow
  - as should be obvious to any linguist, this is analogous to a phoneme
  - example: ⟨x⟩ = {⟨x⟩, ⟨𝑥⟩, ⟨𝗑⟩, ⟨𝐱⟩, ⟨𝓍⟩, ⟨𝔵⟩, ⟨𝕩⟩, …}
  - notice that ⟨x⟩ ≠ {⟨X⟩, ⟨𝑋⟩, ⟨𝖷⟩, ⟨𝐗⟩, ⟨𝒳⟩, ⟨𝔛⟩, ⟨𝕏⟩, …}; this distinction will be explained later
  - often people ignore the distinction between graph and grapheme, but it matters a lot in systems of digital text so keep it in mind through this talk
  - when I give a form within ⟨angle brackets⟩ it represents a grapheme unless otherwise stated
- a SCRIPT is a tradition of writing based around a set of graphemes: e.g. Latin, Cyrillic, Devanagari
- a WRITING SYSTEM (including a TRANSCRIPTION SYSTEM) defines a set of graphemes and a mapping between them and some set of phones, phonemes, syllables, etc.
  - so the IPA defines the grapheme ⟨x⟩ as representing [x] and thus /x/ as necessary
    - English however defines the grapheme ⟨x⟩ as usually representing /ks/
  - Japanese defines the grapheme ⟨か⟩ as representing /ka/ – nobody else uses this grapheme though
  - an ORTHOGRAPHY is a writing system together with predictive and proscriptive rules for spelling, punctuation, annotation, and other paralinguistic phenomena
    - the distinction between 'writing system' and 'orthography' is rarely made in practice
- writing systems are not really that simple – consider ⟨H⟩
  - in English this may be /h/ in ⟨Happy⟩ /ˈhæˌpi/ or sometimes /∅/ as in ⟨An History⟩ /æn‿ˈɪsˌtɹi/
  - it can be part of a phoneme representation too, e.g. ⟨CH⟩ = /tʃ/, ⟨PH⟩ = /f/
    - pairs of graphemes used to represent phonemes are called DIGRAPHS
    - also TRIGRAPHS, TETRAGRAPHS, and even PENTAGRAPHS: German ⟨tzsch⟩ = /tʃ/ in ⟨Nietzsche⟩
  - the same grapheme is used in Greek – ⟨Η⟩ = Ancient Greek /ɛː/, Modern Greek /i/
  - the same grapheme is used in Russian – ⟨Н⟩ = /n/
  - keep this problem in mind, since Unicode is a particular solution to representing this variety digitally
- some graphs and graphemes are special in particular writing systems
  - PUNCTUATION such as ⟨ ⟩ 'space', ⟨,⟩ 'comma', ⟨“⟩ 'double quote', ⟨—⟩ 'em dash', ⟨¿⟩ 'inverted question'
  - DIACRITICS such as ⟨´⟩ 'acute', ⟨¯⟩ 'macron', ⟨˛⟩ 'ogonek', ⟨◌̐⟩ 'chandrabindu', ⟨◌̛⟩ 'horn', ⟨◌̉⟩ 'hook'
  - SYMBOLS such as ⟨$⟩ 'dollar sign', ⟨¶⟩ 'paragraph', ⟨*⟩ 'asterisk', ⟨№⟩ 'numero', ⟨♡⟩ 'heart'
  - we will meet them again later when we discuss Unicode

[*] University of British Columbia; jcrippen@gmail.com

hex	*dec*	*bin*	*hex*	*dec*	*bin*	*hex*	*dec*	*bin*	*hex*	*dec*	*bin*
0	0	0	8	8	1000	10	16	10000	18	24	11000
1	1	1	9	9	1001	11	17	10001	19	25	11001
2	2	10	A	10	1010	12	18	10010	1A	26	11010
3	3	11	B	11	1011	13	19	10011	1B	27	11011
4	4	100	C	12	1100	14	20	10100	1C	28	11100
5	5	101	D	13	1101	15	21	10101	1D	29	11101
6	6	110	E	14	1110	16	22	10110	1E	30	11110
7	7	111	F	15	1111	17	23	10111	1F	31	11111

Table 1: Hexadecimal, decimal, and binary numbers from 0 to $2^4 - 1 = 31$.

2. Bits and bytes

Computers were first invented for numeric processing, and in essence everything else since then has been an

- the bit is the fundamental unit of computing information
  - some would say *all* information, but I'll ignore that issue for this talk
  - a bit can have one of two values 0 and 1; these are exclusive so a bit can be either but not both
  - the symbols ⟨0⟩ and ⟨1⟩ are arbitrary; ⟨−⟩ and ⟨+⟩ would be fine, as would ⟨•⟩ and ⟨∘⟩ or ⟨ω⟩ and ⟨ȸ⟩
- the binary number system is based on the bit
  - mathematical logic using binary numbers is called boolean logic after its inventor George Boole; consequently binary values are often called boolean values by computer programmers
  - binary numbers are based on powers of two per unit; cf. decimal powers of ten per unit
    - the lowest bit b_0 has the value $b_0 \times 2^0$, the next lowest bit b_1 is $b_1 \times 2^1$, the next b_2 is $b_2 \times 2^2$, etc.
    - so the binary number 101 has the value $(\mathbf{1} \times 2^2) + (\mathbf{0} \times 2^1) + (\mathbf{1} \times 2^0) = 4 + 0 + 1 = 5$
- binary numbers are extremely difficult for most humans to read, especially when the values are large
  - other number systems are used as shorthands to represent chunks of binary numbers
    - the two most often encountered are octal and hexadecimal
  - the octal number system is base eight, so each unit is a power of 8 and has values from 0 to 7
  - the hexadecimal[1] system is base 16, so each unit is a power of 16 and has values from 0 to 16
    - octal used to be more popular, but hexadecimal is the most common today
  - hexadecimal units – a.k.a. 'hexits' by analogy with decimal 'digits' – represent more values per unit than decimal units do, so more than 10 symbols (thus graphemes) are needed for them
    - hexadecimal was invented by Latin script users, so one counts from 0 to 9 and then A to F
    - in English they are simply read off sequentially: 1D /wʌn di/, */ˈdiˌtin/
  - because hexadecimal symbols overlap with decimal, there are distinguishing marks for hex
    - nobody agrees on a particular mark; e.g. the C programming language uses a prefix ⟨0x⟩, the Pascal language uses prefix ⟨$⟩, Common Lisp ⟨#x⟩, and Intel assembly a suffix ⟨h⟩ or ⟨H⟩
    - mathematicians and computer scientists often use a subscript number, which I will use here
  - so the binary number 1111_2 is decimal 15_{10} and hex F_{16}
  - the hex number CAFE_{16} is $(12 \times 16^3) + (10 \times 16^2) + (15 \times 16^1) + (14 \times 16^0) = (12 \times 4096) + (10 \times 256) + (15 \times 16) + (14 \times 1) = 49152 + 2560 + 240 + 14 = 51966$
    - each hexit represents four bits: CAFE_{16} = 1100 1010 1111 1110

1. The term 'sexadecimal' would be more properly Latinate, but this invites the taboo abbreviation 'sex' and so has been avoided.

- computers usually do not process bits alone, but instead handle larger chunks of bits all at once
  - a BYTE /bai̯t/ is nowadays usually eight bits, or two hexits: 1111 1110 = FE_{16}
    - historically, byte sizes ranged between six and nine bits depending on the particular computer architecture, but the eight-bit byte has become universal[2]
    - because the size of a byte can vary, an explicitly eight-bitted chunk is called an OCTET
  - bigger chunks of bits that computers work with have more varying names, and are often dependent on particular architectures: 'nybble', 'deckle', 'halfword', 'word', 'dword', 'quadword', etc.
- in working with digital data, it is essential to remember that everything in a computer is binary and hence composed of all bits and bytes
  - digital text is made up of long strings of bits
  - audio is made up of complex multidimensional matrices of bits
  - video is made up of even more complex matrices of bits
  - computer programs are just long strings of operations and data coded in bits
  - the image on a screen is represented by a two-dimensional matrix of triplets (red, blue, green) of bits
- computers do not 'understand'[3] that there are differences between bits used for one purpose or another
  - a computer doesn't really know that some bits represent a picture but others represent a sound
  - instead humans – programmers and users – define and create the contexts in which bits are interpreted
    - these contexts are what give meaning to the bits via the behaviour of computer programs
  - a linguistic analogy is the meaninglessness of speech sounds – they only obtain meaning through their context in a defined linguistic system
- a harsh consequence then follows from a computer's ignorance about the meaning of bits
  - digital data is useless without a program (or knowledgable human!) to contextualize the data usefully
  - by maintaining programs, we prolong the useful life of the data
  - by establishing standards for data, we ensure that different programs contextualize it the same way

3. Characters, character sets, and character encodings

- DIGITAL TEXT means any written linguistic or paralinguistic information represented in a binary code
- the smallest unit of digital text is the CHARACTER
  - the conventional term 'letter' is inappropriate: characters include punctuation, diacritics, symbols, blanks ('whitespace'), formatting instructions, and many other non-letter entities
  - every character must be represented by a unique sequence of bits
    - the bit sequences used have some logic in their systems, but they are mostly arbitrary
    - consider Latin lowercase ⟨a⟩: 0110 0001 = 61_{16} = 97_{10}
  - characters can occur in various sizes, ranging from two bits to 32 bits or more
    - the I Ching (*Yì Jīng*) trigrams use three bits: ☰ ☱ ☲ ☳ ☴ ☵ ☶ ☷
    - Braille is a six-bit system: ⠃ ⠗ ⠁ ⠊ ⠇ ⠇ ⠑ *BRAILLE*
    - Morse code uses a variable number of bits: –– ––– ·–· ··· · *MORSE*
      - most Morse characters are only two or three bits, the maximum is six ·––·–· @
- before the spread of Unicode, the most widely used standard for characters was ASCII
  - American Standard Code for Information Interchange,[4] /ˈæs.ki/

2. The Common Lisp programming language supports variably sized bytes of anything between 1 and 36 bits. This is unusual, and is due to its ancestry on systems with less common byte sizes. The File Transfer Protocol also supports rare byte sizes.
3. There is an old joke in computing: "don't anthropomorphize computers because they don't like it".
4. ASCII is standardized in the US as ANSI X3.4–1986 and its predecessors, and internationally as ISO 646.

  - ASCII is a seven-bit system, so a maximum number of $2^7 = 128$ characters
  - ASCII is sufficient for basic English with limited punctuation, e.g. no en-dash ⟨–⟩ or section sign ⟨§⟩
  - many pre-Unicode eight-bit systems were developed to extend ASCII: Mac Roman, IBM CP 437, Windows-1251, the ISO/IEC 8859-*x* family, etc.
  - a competitor that is now obsolete but still occasionally bumped into was IBM's EBCDIC
    - Extended Binary Coded Decimal Interchange Code, /ˈɛb.si.dɪk/
    - EBCDIC was an eight-bit system, so a maximum number of $2^8 = 256$ characters
    - there were multiple incompatible versions with some bad design decisions, so it died out
- a CHARACTER SET is a collection of characters and the specification for them
  - a character set only defines what the characters should exist in the system, how to present them, and how they should behave in various contexts
  - character sets do *not* specify the binary representation of characters
- a CHARACTER ENCODING (usually just 'encoding') specifies how characters should be represented in binary
  - a similar term is CODE PAGE due to MS-DOS influence
  - the distinction between character set and character encoding is often ignored because they are usually defined in the same standard
  - but when talking about multiple incompatible systems the distinction is useful
    - the character appearing as ⟨a⟩ is 61_{16} in ISO 8859-1, Windows-1252, and Mac Roman
    - in contrast ⟨é⟩ is $E9_{16}$ in ISO 8859-1 and Windows-1252 but $8E_{16}$ in Mac Roman
    - so character sets may cover the same characters, but their encodings may be different
  - Unicode defines multiple encodings for the same character set, so the distinction gets more important
- a REPERTOIRE is the range of different characters available in a character set
  - ASCII has a small repertoire that is essentially only suitable for English
  - ISO 8859-1 has a small repertoire that is useful for western European languages, but little else
  - Unicode has an extremely large repertoire – enough to support nearly all known writing systems
- a CODE POINT is a single element in a specific character set
  - usually a code point is defined as part of a two or three dimensional matrix of binary codes
  - the codes then are implicitly a kind of encoding, but not necessarily the only encoding possible
  - the code point of ⟨é⟩ in the ISO 8859-1 character set is $E9_{16}$ which is also its encoding
  - the code point of ⟨é⟩ in Unicode is $E9_{16}$, but in the UTF-8 encoding it's $C3A9_{16}$
- text in one encoding is often unreadable when processed as though it were in a different encoding
  - this is usually done by accident, but sometimes it's done for entertainment or crude encryption
  - the result is MOJIBAKE, a term taken from Japanese *mojibake* 'character mutation'
  - if we take ⟨文字化け⟩ *mojibake* in UTF-8 and interpret it using ISO 8859-1 we get ⟨æ–‡å—åŒ–ã⟩
    - hardcore digital text nerds can recognize particular kinds of mojibake from encoding patterns
    - the resulting strings of GARBAGE like ⟨Ãƒâ€šÃ‚Â£⟩ are also derisively called UNICRUD
  - other names for the phenomenon include Mandarin Chinese ⟨亂碼⟩ or ⟨乱码⟩ *luànmǎ* 'chaos code', Korean ⟨외계어⟩ *oigyeeo* 'alien language', Russian ⟨кракозябры⟩ *krakozjabry* 'childish scribbles', and German *Zeichensalat* 'character salad'
  - a similar but distinct situation is when the current font lacks characters
    - I am unaware of any terms for this, and only "▢▢▢▢▢" comes to mind
- moving text from one encoding to another is called CONVERSION
  - ONE-WAY CONVERSION is where the conversion cannot be completely reversed
  - ROUND-TRIP CONVERSION is where the text can be converted back and forth without loss
  - Unicode includes some seemingly superfluous or duplicate characters to enable round-trip conversion

- INCOMPLETE CONVERSION is when most but not all of the text is converted successfully
  - DROP-OUTS are missing characters, entirely absent in the result: ⟨drop•out⟩ → ⟨dropout⟩
  - TURDS are encoding elements that survive conversion and appear as superfluous characters
    - ⟨turd : 糞⟩ → ⟨turd : ESC$B糞⟩ — an invisible ISO 2022-JP escape code wrongly survives here
  - DREAD QUESTION MARK DISEASE is the result of incorrectly converting from Windows-1252 to ISO 8859-1, usually caused by using a Microsoft Office product to generate HTML web pages
    - ⟨“Microsoft® Word™”⟩ → ⟨?Microsoft? Word??⟩

4. The Unicode character set

- by the late 1980s, there was a bewildering variety of character set standards in use around the world
  - as well as those already mentioned, there was ANSEL in libraries, KOI8-R for Russian, JIS X 0208 for Japanese, JIS X 0212 for more Japanese, GB2312 for Simplified Chinese, HKSCS for Cantonese, Big5 for Traditional Chinese, KS X 1001 for Korean, VISCII for Vietnamese, TIS-620 for Thai, etc., etc., etc.
  - Unicode was designed to replace this alphanumeric soup with one single standard that could support every single writing system in use worldwide
- Unicode was begun in 1991 though the earliest efforts date to 1988
  - initially the goal was to halt the proliferation of incompatible standards for East Asian languages
  - because they were all derived from Chinese sources, termed the HAN script in Unicode jargon, it was believed they could be unified into a single system called UNIHAN
  - but their goals soon expanded to cover every writing system, not just the East Asian ones
- today the Unicode standard is maintained by the UNICODE CONSORTIUM
  - most members are computer hardware and software companies: Apple, Microsoft, Google, IBM, etc.
  - the national government of India and the University of California Berkeley are also members
  - a couple of linguistic organizations are members: SIL International, Evertype, and Tavultesoft
  - LIASON MEMBERS develop standards that impact the Unicode standard, and this includes the LSA
  - membership in the Unicode Consortium is open to the public, but membership is not useful except for people with an extensive and long term interest in the standardization process
  - the International Standards Organization (ISO) adopted Unicode as the ISO 10646 standard
    - the ISO standard lags behind the actual Unicode Consortium publications, so people ignore it
- Unicode started out as a 16 bit character set with 2^{16} = 65536 characters from 0000_{16} to $FFFF_{16}$
  - the stated aim was to “encompass all the characters of the world’s languages”
  - “in a properly engineered design, 16 bits per character are more than enough for this purpose”
    - they were of course wrong, as even covering all of Chinese proved to be much larger
- today Unicode uses a maximum of 32 bits, but there is more to it than that
  - 2^{32} is about 4 billion, but Unicode specifies a complex structure that limits its size to 1.1 million
    - as of Unicode 6.1 (2012), there are 249,763 code points allocated for some purpose
    - 109,973 code points are graphical and intended for characters representing graphemes
      - 75,619 of these code points are for Han, 11,172 are for Hangul (Korean)
    - 137,468 code points are reserved for Private Use, meaning that the Unicode Consortium will never specify them and thus they can be used outside of the standard without obsolesence
- structure of the Unicode character set
  - the basic division is between different PLANES: large collections of continuous code points
  - the most commonly used plane is the Basic Multilingual Plane (BMP) from 0000_{16} to $FFFF_{16}$
  - the extant planes in Unicode as of version 6.1 are:

000000 … 00FFFF	Plane 1, Basic Multilingual Plane (BMP)
010000 … 01FFFF	Plane 2, Supplementary Multilingual Plane (SMP)
020000 … 02FFFF	Plane 3, Supplementary Ideographic Plane (SIP)
030000 … 0DFFFF	reserved
0E0000 … 0EFFFF	Plane 14, Supplementary Special-Purpose Plane (SSP)
0F0000 … 0FFFFF	Plane 15, Supplementary Private Use Area–A
1F0000 … 10FFFF	Plane 16, Supplementary Private Use Area–B
110000 … FFFFFF	reserved

  - each plane can be divided into blocks which are small continuous sequences of code points
  - blocks in the BMP are small and numerous, usually devoted to specific scripts or purposes
  - the Latin script has the largest number of blocks, but they're all fairly small
  - the largest block is CJK Unified Ideographs Extension B in the SIP, with 42,719 code points
  - here's a nonrandom sample of blocks in the BMP:

0000 … 007F	Basic Latin
0080 … 00FF	Latin-1 Supplement
0100 … 017F	Latin Extended–A
0180 … 017F	Latin Extended–B
0250 … 02AF	IPA Extensions
1D00 … 1D7F	Phonetic Extensions
1E00 … 1EFF	Latin Extended Additional
2700 … 27BF	Dingbats
2C60 … 2C7F	Latin Extended–C
A720 … A7FF	Latin Extended–D

  - within individual blocks there are obvious organizational principles, but they are block-specific
  - not all code points in a block are necessarily assigned to specific characters; often some space is reserved for future additions

- Unicode defines a number of different types of characters
  - graphic characters are what we usually think of as characters
    - not only letters and numbers, but punctuation, weather symbols, smiley faces, mathematical symbols, box-drawing elements, game symbols, and other oddities
    - basically a graphic character is anything that people will see
  - format characters represent invisible formatting commands
    - these are meant to be interpreted by display software that presents text to the reader
    - word-processor things like 'italic' and 'bold' are excluded, they are presentation issues that are properties of the typography and design and not of the text itself
    - typical examples include right-to-left and left-to-right print order indicators, invisible hyphens that indicate where an optional hyphen can be inserted, etc.
  - control code characters are a small set inherited mostly from ASCII
    - they were originally meant to control teletypewriters
    - they're still visible on modern keyboards: carriage return, tab, line feed, backspace, etc.
  - private use characters have reserved, valid code points
    - the Unicode standard requires that programs process them just like other graphic characters
    - Unicode does not specify how they should be displayed, nor how they should act in any context
    - thus private use characters can be used to represent things that aren't standardized
    - linguists should generally consider applying to have characters standardized rather than relying on private use characters, but they are a good stop-gap for new characters
  - surrogate characters are intended for use in the 16 bit UTF-16 encoding of Unicode

        · they have no function alone, but must occur in surrogate pairs
        · I will discuss them when I talk about the UTF-16 encoding later
    – reserved characters are those which aren't assigned yet, but may be in the future
    – noncharacters are specified to not be characters
        · the standard guarantees that they will never be used in future versions, unlike reserved characters
        · for example, $FFFE_{16}$ is a noncharacter 'byte order mark', intended to be used as a means of detecting the order in which bytes are stored in a computer
            › big-endian systems store bytes from largest to smallest, e.g. Power PC
            › little-endian systems store bytes from smallest to largest, e.g. Intel x86
            › if a program expects to find $FFFE_{16}$ in a file but finds $FEFF_{16}$ instead then it knows the current system has the opposite byte order from the one on which the file was created
        · most other noncharacters have similar technical purposes

- the names of characters in Unicode are also standardized, not just their positions
    – code points are represented by U+*xxx…* where *xxx…* is a hexadecimal number
    – most characters have official names, and these are usually written in all uppercase or small caps
    – here are some examples:

· é	U+00E9	Latin Small Letter E With Acute
· ʔ	U+0294	Latin Letter Glottal Stop
· བྷ	U+0F57	Tibetan Letter Bha
· ≆	U+2246	Approximately But Not Actually Equal To
· ㉀	U+3240	Parenthesized Ideograph Festival
· ﬄ	U+FB04	Latin Small Ligature FFL
· 𐃅	U+100C5	Linear B Ideogram B225 Bathtub
· 𐏉	U+103C9	Old Persian Sign Auramazdaa-2
· 𝖂	U+1D582	Mathematical Bold Fraktur Capital W
· 🀗	U+1F017	Mahjong Tile Eight of Bamboos
· 😒	U+1F612	Unamused Face
· 🚶	U+1F6B6	Pedestrian
· 🜟	U+1F71F	Alchemical Symbol For Regulus of Iron

    – most CJKV characters don't have names apparently because they're multilingual and naming them would privilege a particular language
    – the long, verbose names are lovingly referred to as 'Unicrud names' by a few nerds

- graphic Unicode characters do not, with some exceptions, encode font variation
    – this means that Unicode characters represent *graphemes*, not *graphs*
    – mathematical characters are special because the different fonts have meaning
        · so ⟨𝑘⟩ ≠ ⟨𝒌⟩ ≠ ⟨𝓴⟩ ≠ ⟨𝔨⟩ ≠ ⟨𝕜⟩ even though they are all visibly just variant forms of ⟨k⟩
        · for similar reasons, we linguists are happy that they have ⟨a⟩ ≠ ⟨ɑ⟩

- Unicode does encode separate characters for similar graphemes because they have different meanings
    – U+0021 ⟨!⟩ Exclamation Mark ≠ U+01C3 ⟨ǃ⟩ Latin Letter Retroflex Click
    – U+2019 ⟨'⟩ Right Single Quotation Mark ≠ U+02BC ⟨ʼ⟩ Modifier Letter Apostrophe
    – to capture these differences, Unicode specifies character categories and character properties
        · there are a bunch of categories: Lu 'Letter, uppercase', Lt 'Letter, titlecase', Mc 'Mark, spacing combining', Nd 'Number, decimal digit', Pd 'Punctuation, dash', Sc 'Symbol, currency', etc.
        · combining characters, e.g. diacritics, are specified to attach to base letters in different places
        · Brahmi-derived scripts of South and Southeast Asia have complicated reordering rules
        · some scripts have characters that behave differently depending on the writing direction, e.g. changing the direction of opening and closing parentheses

5. Unicode encodings

- text in the Unicode character set can be encoded by using just the code points themselves
  - this is called Unicode Transformation Format 32, abbreviated UTF-32
  - although Unicode specifies 2^{32} bits per character, most characters have only 8 or 10 distinct bits, so U+00E9 ⟨é⟩ is actually U+000000E9 though only the lowest byte is interesting
  - three bytes of space are wasted for every character below FF_{16}, which is egregiously wasteful for most European languages
  - but several Unix-based operating systems do use UTF-32 internally because the algorithm for encoding characters is trivially simple
  - nobody seriously uses UTF-32 to store text because it's so bloated
- the UTF-16 encoding uses only two bytes per character
  - characters with code points below U+FFFF are just represented directly, so all of the BMP
    - our old friend U+00E9 ⟨é⟩ is just $00E9_{16}$ with two bytes, the upper being all zeroes
  - above U+FFFF, the encoding uses SURROGATE PAIRS
    - the leading surrogate is in U+D800 … U+DBFF and the trailing within U+DC00 … U+DFFF
    - so ⟨𐎠⟩ U+103A0 Old Persian Sign A is the sequence $D800_{16}$ $DFA0_{16}$ in UTF-16 rather than just $000103A0_{16}$ in UTF-32
  - for most text using UTF-16 still wastes space but much less so than UTF-32
  - UTF-16 is used internally by Microsoft Windows, Apple Mac OS X, and the Java virtual machine
- UTF-8 is the most space-efficient encoding in wide use
  - for Basic Latin UTF-8 only uses one character, making it identical to ASCII
  - many other characters are encoded in two bytes, and the system extends to three or four bytes as necessary
    - the number of bits per character, the CHARACTER LENGTH, thus varies between different code points (actually between different blocks)
    - here's a breakdown of the character length:

› U+0000 … U+007F	1 byte per character
› U+0080 … U+07FF	2 bytes per character
› U+0800 … U+FFFF	3 bytes per character
› ≥ U+10000	4 bytes per character

  - a subtle but vital feature is that no characters above U+007F are ever represented using characters below that, so that conversion always preserves ASCII characters even if the others get mangled
    - because most text markup languages (XML, HTML, TeX) use ASCII, this preserves their structure
  - the most frequently used code points require the least number of bytes, so the system is very efficient
  - here are some examples of UTF-8 encoding:

· a	U+0061	below U+007F, so 1 byte	→ 61
· α	U+03B1	below U+07FF, so 2 bytes	→ CE B1
· Ѩ	U+0468	below U+07FF, so 2 bytes	→ D1 A8
· अ	U+0905	below U+FFFF, so 3 bytes	→ E0 A4 85
· あ	U+3042	below U+FFFF, so 3 bytes	→ E3 81 82
· 𐏉	U+103C9	above U+FFFF, so 4 bytes	→ F0 90 8F 89

  - UTF-8 is hard to understand but it is so well supported that users don't need to think about it
  - it is by far the preferred encoding for representing digital text today
- Punycode is an encoding in ASCII, used for internationalized domain names in DNS
- there are other weird encodings like UTF-7 and UTF-9/UTF-18 but they're very rare
- there are also old encodings called UCS-2 (≈ UTF-16) and UCS-4 (≈ UTF-32) from ISO which are rare

6. Unicode beyond characters

- most people, even programmers, think of Unicode as just a character set and encodings
  - actually Unicode includes much more than that
  - the standard describes each writing system, so it's a good reference especially for some obscure ones
  - the standard also defines how various types of characters should behave, and this is important to linguists because our expectations may be different from the standard
- combining characters and normalization forms
  - a combining character is a graphic character that is intended to appear on top of a base character
  - combining characters are mostly diacritics, but also include things like surrounding circles
  - combining characters *follow* their base characters; consider the following sequence:

    ⟨e⟩ U+0065 Latin Small Letter E
    + ⟨´⟩ U+0301 Combining Acute Accent
    + ⟨a⟩ U+0061 Latin Small Letter A
    → ⟨éa⟩ always
    ↛ ⟨eá⟩ never
  - so the grapheme ⟨é⟩ is represented by the character sequence U+0065 U+0301
  - conventionally ⟨◌⟩ U+25CC Dotted Circle is used as a host for diacritics so that it's clear where they are positioned on the base character: ⟨◌́⟩, ⟨◌̱⟩, ⟨◌̉⟩, ⟨◌̕⟩, ⟨◌̸⟩, ⟨◌ͪ⟩, ⟨◌ʰ⟩
  - there is no standard limit on the number of combining characters
    - in practice more than a couple will be illegible or simply fail to display
    - nonetheless, the character data will still be functional for text processing purposes
  - recall that Unicode also specifies ⟨é⟩ U+00E9 Latin Small Letter E With Acute
    - this is a precomposed character that can be decomposed into a sequence of characters
    - the decomposition of precomposed characters is standardized
  - why have precomposed characters at all?
    - mostly for historical purposes, to support round-trip encoding between other systems
      - the canonical example is compatibility with VISCII, and thus all those Vietnamese vowels
    - in some cases, it's hard for display systems to handle the diacritics: ⟨ⱡ⟩ U+2C61 Latin Small Letter L With Double Bar, ⟨ⱦ⟩ U+2C66 Latin Small Letter T With Diagonal Stroke
      - these sorts of characters *do not* have a decomposition specified so that people aren't tempted to represent them using combining characters
    - in general the Unicode Consortium prefers to handle new Latin characters using combining characters when possible, since most new Latin characters are actually extensions of the basic set with diacritics; this helps reduce hidden complexity and simplifies adding new Latin characters
      - it's not Unicode's problem that you can't find a font to display some combination correctly — this is a presentation problem in their terminology, and you need to bug font designers and/or operating system developers to handle your character properly
      - a constant bugaboo for me is ⟨◌̱⟩ U+0331 Combining Macron Below, e.g. ⟨x̱⟩ and ⟨g̱⟩
        - Unicode just specifies this like any other combining character
        - but Microsoft has been terribly slow at supporting it correctly in all their software
        - *this is not Unicode's fault* — it's Microsoft's fault for not properly implementing the standard
  - Unicode text may be a random mix of precomposed characters and combining + base characters
    - this is called denormalized and fixing it is a process called normalization
    - Unicode specifies two normalization forms for normalization of characters
      - Normalization Form Canonical Decomposition (NFCD) has all precomposed characters decomposed to combining + base character sequences

      - › Normalization Form Canonical Composition (NFCC) has all possible combining + base sequences replaced with precomposed characters
    - · NFCC is notably preferred by Mac OS X, and NFCD is used by Windows and Linux
    - · note that only potentially precomposed characters are affected by normalization
      - › since ⟨v́⟩ does not have a precomposed character assigned to it, the representation will *always* be a sequence of ⟨v⟩ U+0076 Latin Small Letter V + ⟨´⟩ U+0301 Combining Acute Accent
  - – Unicode specifies that it mostly doesn't care about the order of combining characters, at least for Latin
    - · that doesn't mean that multiple combining characters will always work in any order
    - · instead, it's largely up to the display system programmers and the font designers to make various orders of combining characters work properly
    - · this means that you may occasionally find combinations that break sometimes
      - › so ⟨V̰́⟩ = ⟨V⟩ + ⟨◌́⟩ U+0301 Combining Acute Accent + ⟨◌̰⟩ U+0330 Combining Tilde Below
      - › or ⟨V̰́⟩ = ⟨V⟩ + ⟨◌̰⟩ + ⟨◌́⟩ — Unicode doesn't care
      - › in fact, using the Cambria font (v. 5.97) in XƎLaTeX the former with ⟨◌́⟩ first produces ⟨V̰́⟩
      - › but the latter with ⟨◌̰⟩ first surprisingly produces ⟨V̰́⟩, which we obviously don't want
      - › so if you have a combination that breaks, try switching their order
      - › if the order of diacritics turns out to be your problem, file a bug report with the font designers and/or with the display software developers
- case – ⟨a⟩ versus ⟨A⟩
  - – for hundreds of years, printing was done with small metal type blocks, called 'type'
    - · types were stored in large wooden boxes or drawers called 'typecases' or just 'cases'
    - · a full set of a single size of a single typeface (font) came in two cases, one with the capital (majuscule) letters and punctuation, and the other with the small (miniscule) letters
    - · during use, the capital case was put in the upper rack and the other in the lower rack: thus uppercase and lowercase, which persist to this day
  - – so case pairs two graphemes in a mapping, e.g. ⟨a⟩ ↔ ⟨A⟩ even though ⟨a⟩ ≠ ⟨A⟩
  - – several scripts distinguish case – Latin, Cyrillic, Greek, Coptic, Glagolitic, Armenian, archaic Georgian
  - – in Unicode case is actually two features: [±lower] and [±upper]: ⟨h⟩ [+l, −u], ⟨H⟩ [−l, +u], ⟨'⟩ [−l, −u]
  - – Unicode also specifies rules and algorithms for how to transform between cases
    - · this means that programs can automagically convert between uppercase, lowercase, titlecase (initial letters uppercase), small caps, small caps + titlecase, iNVERSE titlecase, etc.
  - – linguists often make implicit, unconsidered assumptions about case when designing orthographies with case-sensitive alphabets like Latin and Cyrillic
    - · Unicode already specifies case pairings between e.g. ⟨H⟩ and ⟨h⟩
    - · you might like ⟨Ḧ⟩ to pair with ⟨h̤⟩ but in fact you can't do that; the standard dictates ⟨ḧ⟩ instead
    - · and requiring your users to maintain the nonstandard pair ⟨Ḧ⟩ ↔ ⟨h̤⟩ means that they will have to fight with all kinds of software from now until Unicode is obsolete (i.e. forever)
    - · you can provide *fonts* that present ⟨h̤⟩ instead of ⟨ḧ⟩, but then your users are tied to the use of specific fonts; for Latin and Cyrillic-based orthographies this is an artificial limitation that people will quickly become annoyed with, and they will come to hate you for restricting them
  - – it's possible to have unusual case pairings standardized by Unicode, but this is an arduous process that will affect many things you've never thought of; it's just best to go with the existing standards
  - – there are a few precomposed characters that lack case pairs, e.g. ⟨ẘ⟩ U+1E98 Latin Small Letter W With Ring Above
    - · the uppercase form can only be a base + combining character pair: ⟨W⟩ + ⟨◌̊⟩
    - · for sanity's sake, linguists designing orthographies should try to avoid these case-orphans so that trouble with them won't crop up in the future

- sorting and collation
  - collation is the official Unicode term for what most programmers call sorting: "the process of ordering units of textual information"
  - not all Latin-based writing systems are ordered in the same way
    - Swedish ends its alphabet with ⟨… X Y Z Å Ä Ö⟩, but Danish & Norwegian have ⟨… X Y Z Æ Ø Å⟩
    - most German dictionaries ignore the differences between ⟨A⟩ and ⟨Ä⟩ for example, but in telephone directories ⟨Ä⟩ = ⟨AE⟩ ≠ ⟨A⟩
    - many computer programs sort uppercase before lowercase even though users don't think of them as being sorted separately
  - new collation sequences can be added to the standard, but this takes a lot of time
  - ideally the default Unicode Collation Algorithm should be adopted, but sometimes this won't work
  - linguists are better off adopting collation sequences used by local majority languages because they will be the defaults in most software in the area
  - the worst idea is declaring a collation sequence that is unsupported by software; users will hate you

7. Practical issues with Unicode

- choose the right character for the job!
  - the standard specifies different semantics for characters that "look the same"
    - ⟨A⟩ U+0041 Latin Capital Letter A ≠ ⟨Α⟩ U+0391 Greek Capital Letter Alpha ≠ ⟨А⟩ U+0410 Cyrillic Capital Letter A
    - mixing these will cause subtle problems because software expects them to be different
    - depending on the mix of fonts used to display text, they appear very different: ⟨A⟩ versus ⟨A⟩
      - › to emphasize this again, *Unicode doesn't specify appearance*
      - › Unicode only gives a general guide of how characters are supposed to be displayed
      - › the actual glyph that is used for a character depends on the font, the whims of the font designer in approximating various graphs, the display software, and the medium in which the text is displayed (paper, screen, signs, lasers, etched metal, etc.)
        - ∘ by the way, a glyph is a graph implemented in a font, so glyph ⊆ graph
      - › Remember: grapheme ≠ character ≠ glyph!
  - Unicode often provides guidelines on how to use particular characters
    - the following quote is from the Phonetic Extensions Block (U+1D00 … U+1D7F):

      The small capitals, superscript, and subscript forms are for phonetic representations where style variations are semantically important. For general use, use regular Latin, Greek, or Cyrillic letters with markup instead.
    - so ⟨ᴀ⟩ U+1D00 Latin Letter Small Capital A is not meant for ordinary small caps, but instead only as a phonetic symbol (mostly the predecessor of today's IPA ⟨ʌ⟩)
    - small caps in gloss abbreviations and the like should be done using a typographic or word-processor mechanism rather than Unicode characters
- the working linguist's worst offense is using the wrong character for the null symbol
  - ⟨Ø⟩ and ⟨ø⟩ are *letters*, namely U+00D8 and U+00F8 Latin {Capital, Small} letter O With Stroke
  - ⟨ϕ⟩ is U+03D5 Greek Phi Symbol and ⟨φ⟩ is U+03C6 Greek Small Letter Phi
  - the character Unicode wants you to use is ⟨∅⟩ U+2205 Empty Set:

    2205 ∅ EMPTY SET
      = null set
      • used in linguistics to indicate a null morpheme or phonological "zero"

        → 00D8 Ø latin capital letter o with stroke
        → 2300 ⌀ diameter sign

  – each comment that the standard includes is important, and has implications for how the character is processed across systems
    · the first with ‘=’ means that its equivalent name is ‘null set’
    · the second with ‘•’ is a note providing usage information
    · the following two are *contrastive* cross references to other characters that must be *distinguished* from U+2205
  – looking at ⟨ø⟩ U+00F8 Latin Small letter O With Stroke we see
        • Danish, Norwegian, Faroese, IPA
  – that means it’s allowed for use in e.g. phonetic transcriptions, but not as a symbol
  – “but it looks right, so why should I care?”
    · because your text is lying – what you *mean* and what you *write* are then different
    · people can’t search for U+2205 if you’re using U+00F8 instead
    · automatic uppercasing your U+00F8 will change it to U+00D8 and then you’ll be confused and upset
  – “but it looks ugly!” – use a different font!
    · Charis SIL has ⟨∅⟩ which is indeed ugly, more suited to mathematics than linguistics
    · but Cambria has a reasonable ⟨∅⟩ which looks more like what linguists are used to
    · and even better, STIX Variants has ⟨∅⟩, and Minion Pro has ⟨∅⟩

• one other common misuse by linguists is punctuation instead of modifier letters for orthographies
  – Unicode says that punctuation is different from letters, so that software treats them differently
  – so ⟨ʼ⟩ U+02BC Modifier Letter Apostrophe and ⟨ʻ⟩ U+02BB Modifier Letter Turned Comma are real *letters* (category Lm)
  – but e.g. ⟨’⟩ U+2019 Right Single Quotation Mark is punctuation instead
  – Unicode makes this clear in the comments on the former characters

    02BC ʼ MODIFIER LETTER APOSTROPHE
        • glottal stop, glottalization, ejective
        • many languages use this as a letter of their alphabets
        • used as a tone marker in Bodo, Dogri, and Maithili
        • 2019 is the preferred character for a punctuation apostrophe
        → 0027 ' apostrophe
        → 0313 ̓ combining comma above
        → 0315 ̕ combining comma above right
        → 055A armenian apostrophe
        → 07F4 nko high tone apostrophe
        → 1FBF ᾿ greek psili
        → 2019 ’ right single quotation mark