Representing text on the computer: ASCII, Unicode, and UTF-8

STAT/CS 287

James P. Bagrow

Question: Computers can only understand numbers. In particular, only two numbers, 0 and 1 (binary digits or bits). So how do we get computers to read and write text?

Originally, computers (like in the 1950s) didn't bother with this problem, they just expected their operators to read binary data. But eventually this had to be solved.

The solution: determine a way to map each letter/symbol/character to a string of **bits**.

- Character = smallest possible component of a text.

With n bits you have 2^n unique combinations so we can represent 2^n different characters. Then, $n = 7$ gives 128 symbols. This is enough to store the basics of the Roman alphabet (26 lower case letters + 26 upper case letters + 10 digits + punctuation, plus computer control codes like 'end of file', 'start of file', etc.) There are 95 printable characters and 33 non-printing control codes.

However, the mapping from bits to symbols is arbitrary, it's just a table. To exchange data between systems, they need to understand each other's mapping. A standard was needed: ASCII.

- ASCII = American Standard Code for Information Interchange.

ASCII lookup table: character → ASCII CODE (number) → binary representation

In some sense, the ASCII code *is* the character, in that we are not concerned with the *glyph* of the character, its visual representation. An operating system or software toolkit for graphical interfaces usually deals with the problem of taking a character and representing it visually. Excerpt:

Letter	ASCII Code	Binary	Letter	ASCII Code	Binary
a	097	01100001	A	065	01000001
b	098	01100010	B	066	01000010
c	099	01100011	C	067	01000011
.	.	.	.	.	.
.	.	.	.	.	.
.	.	.	.	.	.

- The table is arbitrary but the designers of ASCII settled on some clever ideas. For example, the upper and lower case versions of a letter are always separated by 32, so flipping a single bit will change from upper- to lowercase.
- The process of mapping from the actual character to the binary representation (01100001) is called **"encoding"**. Mapping back from the binary to the character is called **"decoding"**.

A text file is just a long sequence of 8-bit blocks. A program reads them 8 bits at a time and then determines the corresponding letters.

Notice that each binary representation is 8 bits or one byte. $2^8 = 256$ possible numbers. But ASCII defines 128 characters/symbols (0-127) and the first bit of every byte is zero. Why?

ASCII predates many things, including the internet, including the definition of bytes as 8 bits. Many people argued at the time for 7-bits as a "word" and computers were optimized to process data in 7-bit chunks. This is why it's 128 symbols and 7 bits. Eventually, people settled on 8 bits to a byte, optimized their hardware accordingly, and decided to make ASCII backwards compatible by sticking a zero on the front. Problem solved.

ASCII's 128 symbols don't leave much room for nice-looking text like left-vs-right quotes, long and short dashes, and language-specific symbols like diacriticals (umlauts). North Americans didn't care too much but Europeans, Russians, etc. did. So they started using the new 128 symbols made available by putting a 1 at the beginning of each byte. But every country developed their own encodings for what characters 128-255 mean, and reading the text file with the wrong lookup table you will read the file incorrectly, leading to garbage.

Eventually a new standard was settled on for these characters called "latin-1" which standardized somewhat the next 128 symbols, but this doesn't solve the real problem:

- 256 symbols is nowhere near enough space to store all the world's languages.

What happens when, say, an **inter**national **net**work spanning the entire world shows up, and we want to communicate with everyone?

Enter Unicode

A non-profit, the Unicode Consortium, was established in 1991 to make a new lookup table for EVERYTHING. Miraculously, over 25 or so years, they have built a standard table for characters.

The Unicode table consists of 17 "planes" each of which can represent 65,536 $\left(2^{16}\right)$ characters, giving a total of 1,114,112 possible characters.

- Most of Unicode is empty. 136,690 characters have been assigned as of ver. 10.0 (Jun 2017) or 12.27%.

There is likely enough space to represent any character in any language (including extinct languages, fictional languages, emoji) in existence.

Like ASCII, there's some cleverness in how the table is organized. In fact, the original ASCII "numbers" are the same in Unicode. Backwards compatible!

The 1.1 million "slots" in the Unicode table are called "code points" (or sometimes "ordinals"). We can think of them as just numbers. But what Unicode does not do is character encoding—How to map those code points to, e.g., bits and bytes????

Encoding Unicode

There was a long fight over how best to encode Unicode. In fact, much of the Unicode design (its planes) is based around an intended encoding called UTF-16 (UTF = Unicode Transformation Format)

The "obvious" way to encode Unicode is just more bits. $n = 24$ gives $2^{24} = 16.7$ million, plenty of room. Problems with this approach:

1. Since Unicode is backwards compatible with ASCII, the letter A, for example, is now going to be a whole mess of zeros followed by a few 1s and zeros all the way at the right. And this happens for every single ASCII character. Wasteful, it automatically makes every English-language plain text file 3 times bigger (for n = 24).
   - Have to get rid of all the zeros in English text.
2. Historically, 8 zeros in a row is a special computer control code representing the end of the string of characters. So if you send a file with all those leading zeros, many computers will assume the text terminates before it does!
   - Cannot send 8 zeros in a row, anywhere within a valid character.
3. Breaks backwards compatibility. If you are reading plaintext English stored in this new encoding with old code that only understands ASCII, you ideally want it to still be read correctly.

UTF-8, a not-so-obvious encoding, avoids all of these problems!!!

UTF-8

Brilliant idea by Ken Thompson (one of the creators of UNIX).

UTF-8 is a type of **multibyte encoding**. Sometimes you only use 8 bits to store the character, other times, 16 or 24 or more. The challenge with a multibyte encoding is how do you know, for example, that these 16 bits represent a single two-byte character and not two one-byte characters. UTF-8 solves this character boundary problem!

First, if you have a Unicode codepoint under 128 (which is ASCII), you record a zero and then the seven bits of ASCII. All ASCII is automatically UTF-8!

Now, what if we have a codepoint > 128. First, use `110` as a "header" to indicate the start of a new character of two bytes (the two ones tells us it's two bytes). Then the second byte begins with a `10` to indicate "continuation". This leaves 5+6 blank spaces in the remaining bits of these two bytes:

```
110xxxxx | 10xxxxxx
```

(I'm using the "`|`" (space-pipe-space) to visually represent the byte boundaries and `x` to represent a placeholder for any value of a bit)

The remaining blank spaces can then encode the characters: 11 bits or $2^{11} = 2048$ characters.

What about higher codepoints? Start with a header of `1110` instead of `110`. This tells us there are *three bytes* in the character (because there are three ones). The next two bytes again begin with `10` to indicate continuation""

```
1110xxxx | 10xxxxxx | 10xxxxxx
```

16 bits remain to represent characters.

(It's important to use the header to count how many bytes are in the current character so that you can detect if a byte is missing: "Hey, I saw three ones in the header but there's only one continuation before the next header...")

Want more? You can go all the way to:

```
    1111110x | ...
```

or six bytes, which is . . . a *bunch* of possible characters! You actually only need 1–4 bytes to represent all of Unicode ($2^7 + 2^{11} + 2^{16} + 2^{21} = 2,164,864$ characters):

Number of bytes	Bits available for code point	Byte 1	Byte 2	Byte 3	Byte 4
1	7	0xxxxxxx			
2	11	110xxxxx	10xxxxxx		
3	16	1110xxxx	10xxxxxx	10xxxxxx	
4	21	11110xxx	10xxxxxx	10xxxxxx	10xxxxxx

Benefits of UTF-8

- UTF-8 is backwards compatible with ASCII.
- It avoids waste.
- It never sends 8 zeros in a row.
- Most importantly, and why UTF-8 is (finally) the winning encoding: You can read through the string of bits very easily. You don't need an index tracking which character's bytes you are at. If you are halfway through the string and you want to go back to the previous character, you just scan backwards until you see the previous header.

In 2008, UTF-8 became the most popular encoding in the world, surpassing pure ASCII, according to Google. As of October 2019, **UTF-8 accounts for 94.1% of web pages**.

Python and Unicode

https://docs.python.org/3/howto/unicode.html

With this history lesson we've actually addressed one of the challenges with working with Unicode text: the error messages.

- `UnicodeDecodeError` — Something is wrong **reading** the binary data into codepoints that python works with
- `UnicodeEncodeError` — Something is wrong **writing** the code points into binary data

`UnicodeEncodeErrors` often happen when you intend to write Unicode but you've accidentally told Python to only allow plain ASCII. When it encounters a non-ASCII character, you get an error:

```
UnicodeEncodeError: 'ascii' codec can't encode character '\ua000' in
  position 0: ordinal not in range(128)
```

How to read:

- `position 0` - first character is bad
- `ordinal not in range(128)` - our code point is not in [0,127]
- `ascii codec` - our encoding standard is called a ‘codec’ and we chose ASCII (“codec” = coder-decoder)
- `\ua000` - a two-digit hexadecimal encoding of the Unicode code point. Sometimes these are written as `U+xxxxx` (hexadecimal is base-16 that’s why there are letters in the digits)

Reading and writing:

The file handler function in Python supports more arguments than just the filename and the mode (r,w etc)

```
f = open(fname, 'r', encoding='utf-8', errors='strict')
```

`encoding` — tell Python what encoding standard we want to use:

- `ascii` - historical format, supports 128 characters
- `latin-1` - historical format standardized after ASCII went from 7-bit bytes to 8-bit bytes
- `utf-8` - the one true encoding
- `...`

`errors` — what to do if Python encounters an undecode-able byte in the file:

- `'strict'` - raise a ValueError error (or a subclass)
- `'ignore'` - ignore the character and continue with the next
- `'replace'` - replace with a suitable replacement character; Python will use the official U+FFFD REPLACEMENT CHARACTER for the builtin Unicode codecs on decoding and ’?’ on encoding.
- `'surrogateescape'` - Replace with private code points U+DCnn.
- `'xmlcharrefreplace'` - Replace with the appropriate XML character reference (only for encoding).
- `'backslashreplace'` - Replace with backslashed escape sequences.
- `'namereplace'` - Replace with `\N{...}` escape sequences (only for encoding).

See the Python Unicode HOWTO for more.