Things you (probably) didn't know about Unicode

Zaki Mughal

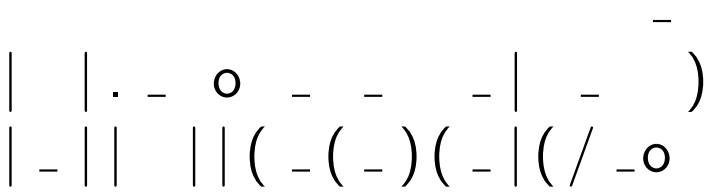
University of Houston:
CougarCS

2013 Oct 10


SautÃƒÂ©ed Ratatouille
Garden fresh eggplant and zucchini with garlic, basil & oregano
Under 5
grams of fat
Calorie Counter
100 calories or less
Sodium Smart
400 mg or less
Nutritional Information
Portion Size: 4 ozw
Calories: 60
Total Fat (g): 2.5
Saturated Fat (g): 0
Sodium (mg): 250
Cholesterol (mg): 0
Carbohydrates (g): 8
Dietary Fiber (g): 2
Protein (g): 1

This is mojibake.

ASCII you a question



- *Unicode* is a standard for encoding, representation, and text handling. It tries to cover every character of the world's languages.

- *Unicode* is a standard for encoding, representation, and text handling. It tries to cover every character of the world's languages.
- It stores each character as a code-point $\rightarrow$ from `U+0000` to `U+10FFFF`. 1,114,112 code-points.

Myths

- Myth: ASCII is good enough. English is the only language used on computers.

Myths

- Myth: ASCII is good enough. English is the only language used on computers.
- We're in Houston. We know this.

Myths

- Myth: ASCII is good enough. English is the only language used on computers.
- We're in Houston. We know this.
- I just needed to get that out of the way.

- Myth: We're using English. We can still use plain text with good ol' ASCII.
- Words you can't store in ASCII: résumé, naïve, noël.

- Myth: We're using English. We can still use plain text with good ol' ASCII.
- Words you can't store in ASCII: résumé, naïve, noël.
- Oh, and ASCII is actually just the characters 0 – 127. The rest (128-255) are extensions that can vary by encoding.

- Myth: We're using English. We can still use plain text with good ol' ASCII.
- Words you can't store in ASCII: résumé, naïve, noël.
- Oh, and ASCII is actually just the characters 0 – 127. The rest (128-255) are extensions that can vary by encoding. There are many encodings.

Myths

- Myth: OK, everything is in Unicode. We'll just use Unicode

Myths

- Myth: OK, everything is in Unicode. We'll just use Unicode
- not everything is in Unicode

Myths

- Myth: OK, everything is in Unicode. We'll just use Unicode
- not everything is in Unicode
- You can't just *use* Unicode. It's a bit harder than that...

Myths

- Myth: OK, everything is in Unicode. We'll just use Unicode
- not everything is in Unicode
- You can't just *use* Unicode. It's a bit harder than that...
- *Languages are hard.*

Myths

- Myth: OK, everything is in Unicode. We'll just use Unicode
- not everything is in Unicode
- You can't just *use* Unicode. It's a bit harder than that...
- *Languages are hard.* In fact, anything dealing with the real world data is *hard*. See: date-time libraries.

Doing things wrong means others will have to try to clean up your mistakes.

Doing things wrong means others will have to try to clean up your mistakes.

- You can't assume encodings.

- You can't assume encodings.
- If you assume an encoding for any reason (ASCII, platform-specific, English-only input), your code is broken.

- You can't assume encodings.
- If you assume an encoding for any reason (ASCII, platform-specific, English-only input), your code is broken.
- This is why you set Content-Type in your HTTP headers or use a `<meta>` in your HTML.

- You can't assume encodings.
- If you assume an encoding for any reason (ASCII, platform-specific, English-only input), your code is broken.
- This is why you set Content-Type in your HTTP headers or use a `<meta>` in your HTML. Or you'll leave browsers guessing. Which you don't want.

- Characters aren't 8-bits.

- Characters aren't 8-bits.
- Yes, `char` in C is still 8-bits.

- Characters aren't 8-bits.
- Yes, `char` in C is still 8-bits.
- A `char*` can not handle Unicode by itself.

- Characters aren't 8-bits.
- Yes, `char` in C is still 8-bits.
- A `char*` can not handle Unicode by itself.
- You need a library like International Components for Unicode (`libicu`) `http://www.icu-project.org/`

How many bits do we need then? (serialization)

- UTF-32: 32-bits, uses most space. Not very useful unless you're an Elf or Klingon.
- UTF-16: 16-bits, a bit of a trade-off, but this and UTF-32 might require endianness handling — BOM (byte-order mark)
- UTF-8: 8-bits, developed by the Plan 9 people at (particularly Ken Thompson). More complex encoding, self-synchronising. Backwards compatible with ASCII

How many bits do we need then? (serialization)

- UTF-32: 32-bits, uses most space. Not very useful unless you're an Elf or Klingon.
- UTF-16: 16-bits, a bit of a trade-off, but this and UTF-32 might require endianness handling — BOM (byte-order mark)
- UTF-8: 8-bits, developed by the Plan 9 people at (particularly Ken Thompson). More complex encoding, self-synchronising. Backwards compatible with ASCII (as long as you don't start making too many assumptions)

You'll need to throw away a lot of assumptions you make about text...

Uppercase and lowercase

```
System.out.printf("%s -(UC)-> %s\n", "\u03c3", "\u03c3".toUpperCase());
System.out.printf("%s -(UC)-> %s\n", "\u03c2", "\u03c2".toUpperCase());
System.out.printf("%s -(LC)-> %s\n", "\u03a3", "\u03a3".toLowerCase());
```

```
σ -(UC)-> Σ
ς -(UC)-> Σ
Σ -(LC)-> σ
```

You can't uppercase every letter:

- The German eszett (ß) has no uppercase equivalent (well, officially)
- If you use all-caps, you need to convert it to two letters: groß→ GROSS.

```
uc( lc( uc(
   "\N{LATIN SMALL LETTER SHARP S}" )))
   eq "SS"
```

You can't uppercase every letter:

- The German eszett (ß) has no uppercase equivalent (well, officially)
- If you use all-caps, you need to convert it to two letters: groß→ GROSS.

```
uc( lc( uc(
   "\N{LATIN SMALL LETTER SHARP S}" )))
   eq  "SS"
```

- Yes, all-caps changes the length of the string...

- Myth: Unicode is just like ASCII but with more characters.
- The standard also specifies algorithms, including:

- casemapping, casefolding
- grapheme clusters
- normalization
- collation
- word- and line-breaking
- properties database (uppercase, lowercase, letter, number, etc.)
- bidirectional text
- glyph variants

These algorithms affect anything you have that deals with text:

- word wrapping
- sorting
- rendering
- changing the case
- sorting strings
- checking if two strings are equal
- regex (character classes)

- Collation: sorting differs from language to language

- Collation: sorting differs from language to language
- German treats ‘o’ and ‘ö’ as the same letter, but Turkish treats them as different letters.

- Collation: sorting differs from language to language
- German treats ‘o’ and ‘ö’ as the same letter, but Turkish treats them as different letters.
- `strcmp` in C compares bytes. `strcoll` uses your *locale*
- on POSIX systems, you can use `LC_*` environment variables to set your locale

- Collation: sorting differs from language to language
- German treats ‘o’ and ‘ö’ as the same letter, but Turkish treats them as different letters.
- `strcmp` in C compares bytes. `strcoll` uses your *locale*
- on POSIX systems, you can use `LC_*` environment variables to set your locale
- if you don’t care about locales, you can set `LC_ALL="C"` and speed up some some utilities like `sort`

- Regex:
- What's a number? `[0-9]`?
- What's a word character `[A-Za-z]`?
- Are glyphs made up of code points combined with diacritical marks equal to their precomposed equivalents?
  é could either be the single code-point U+00E9 or the code-point U+0065 (e) combined with the acute accent code-point U+0301 (´).

- Regex:
- What's a number? `[0-9]`?
- What's a word character `[A-Za-z]`?
- Are glyphs made up of code points combined with diacritical marks equal to their precomposed equivalents?
  é could either be the single code-point U+00E9 or the code-point U+0065 (e) combined with the acute accent code-point U+0301 (´).
  the normalization algorithm converts code-point sequences to a canonical form

- Best source to read about all this is the standard.
- I have some more links that I'll post with these slides.