8 Character encodings

Key terms: `bytes` raw UTF-8 Unicode

Reading: World Wide Web Consortium's "Character encodings: Essential concepts"[17]

Reading: Python Unicode HOWTO[18]

Exercise: Write a program that accurately guesses whether a specified UTF-8 text file is written in Italian or Japanese.

It may seem obvious that an 'L' is only and exactly an 'L', but at a low level we rely on rules for describing and recognizing it. Visually we expect it to resemble the hands of an analog clock showing 3:00. In storage it is composed of a sequence of binary digits (bits), often eight of them, or one byte. Many binary character encodings are available, of fixed or variable width, all of which are designed to serve certain written languages. Historical examples which include Morse Code (variable width and English-based), ASCII (fixed width and English-based), and ISO-8859-1 (fixed width and Western European-based).

The modern standard is UTF-8, which is variable width and interlingual, encoding the Unicode Consortium's character database. Happily, we can now mix as many languages as we want — as long as we have terminals and fonts that support them. There are a few transliteration tools out there (the best may be the International Components for Unicode). Python version 3 strings contain Unicode codepoints, which are stored internally in a custom way for optimization, but expressed and loaded by default as UTF-8.

```
# Determine whether or not the file passed is encoded as UTF-8:

import sys

def check(file):
 with open(file) as handle:
  try:
   handle.read()
   return True
  except UnicodeDecodeError:
   return False

print(check(sys.argv[1]))
```

To describe the information that travels over a network or to and from a file more closely we have `bytes` objects, a rawer form of data that does not necessarily represent characters. Content is decoded according to the mode with which a file is opened, which can be given in the `encoding=` argument to `open()`; binary mode, `mode='rb'`, acquires a `bytes` object rather than a decoded string; this is needed for working with non-character-based "binary" files.

`str.encode()` makes bytes from a string while `bytes.decode()` does the opposite. Therefore, for a string `s`, `s == s.encode().decode()`. Is the inverse true? Not necessarily, because UTF-8 is a variable width encoding, so some series of bytes are not valid. Both these methods will take an `errors=` argument where the policy for encoding and decoding errors can be `ignore` or `replace`.

The strings 'a' and 'á' both contain one codepoint, but UTF-8 requires two bytes to encode the latter: `len('á'.encode('utf8'))> len('a'.encode('utf8'))`. Similarly, "raw" strings without the default backslash interpretation are written `r'hello'`; note that `len(r'\n')> len('\n')`.

Next week, we will address serialized data encodings.

[17] `https://www.w3.org/International/articles/definitions-characters/`
[18] `https://docs.python.org/3/howto/unicode.html`