COMP10001 Foundations of Computing
Character Encoding

Semester 2, 2016
Chris Leckie

October 2, 2016

Lecture Agenda

- Last lecture:
  - Iterators
- This lecture:
  - Functions as arguments to functions
  - Character encoding and multilingual text

Lecture Outline

Functions as Objects

- You may have picked up on the fact that functions in Pythons are "just" objects:

```
>>> min([2,3,1,5])
1
>>> cmp
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'cmp' is not defined
>>> cmp = min
>>> cmp([3,5,1,2,0,-1,9])
-1
>>> cmp2 = min
>>> cmp == cmp2
True
```

Functions as Function Arguments I

- It follows from this that it should be possible to pass functions as arguments to other functions:

```
>>> def minmax(lst, cmp):
...     return cmp(lst)
...
>>> minmax([3,5,1,2,0,-1,9], max)
9
>>> minmax([3,5,1,2,0,-1,9], min)
-1
```

Functions as Function Arguments II

- It further follows that we should be able to use keywords for arguments which are functions:

```
>>> def minmax(lst, cmp=min):
...     return cmp(lst)
...
>>> minmax([3,5,1,2,0,-1,9])
-1
>>> minmax([3,5,1,2,0,-1,9], max)
9
```

Functions as Function Arguments III

- Note the importance of calling the function by the keyword in the body of the function:

```
>>> def minmax(lst, cmp=min):
...     return min(lst)
...
>>> minmax([3,5,1,2,0,-1,9])
-1
>>> minmax([3,5,1,2,0,-1,9], max)
-1
```

Lecture Outline

1. Functions as Arguments to Functions
2. Character Encoding and Multilingual Text

Internal Representation of Characters

- Earlier in the subject, you were introduced to the notion that characters are internally just (positive) integers:

```
>>> ord('a')
97
>>> ord('ö')
246
```

- These values the "code point" values for each character, as based on the Unicode standard

Unicode I

- Unicode is an attempt to represent all text from all languages (and much more besides) in a single standard
- Unicode is intended to support the electronic rendering of all texts and symbols in the world's languages
- Each **grapheme** is assigned a unique number or **code point**
- There is scope within unicode for both **precomposed** (e.g. *á*) and **composite** characters/**glyphs** (e.g. $\acute{} + a$)

Unicode II

- There are plenty of code points to go around (over 1M), to cater for the "big" orthographies
- The code points are split across multiple "planes" of varying size, with the most important being the "BMP" (Basic Multilingual Plane)
- With Unicode, different orthographies can happily co-exist in a single document
- The basic philosophy behind Unicode has been to (monotonically) add more code points for different "languages", starting with the pre-existing encodings

How are Text Documents Represented?

- Text documents are represented as a sequence of numbers, meaning that one possible document representation would simply be the sequence of Unicode code point values of the component characters, e.g.:

```
>>> [ord(i) for i in "computing"]
[99, 111, 109, 112, 117, 116, 105, 110, 103]
```

  meaning that the document containing the single word `computing` could be "encoded" as:

  99111109112117116105110103

  ... or could it?

Text Document Encoding v1 I

- The simplest form of text encoding is through fixed "precision", i.e. a fixed number of digits to represent the code point for each character, e.g. assuming that the highest code point were $10^6 - 1$, we could encode each code point in our document with 6 decimal digits, as follows:

```
000099000111000109000112000117000116000105000110000103
```

Text Document Encoding v1 II

- Given knowledge of the precision, decoding the document would consist simply of reading off 6 digits at a time and converting them into a code point:

```
>>> doc = "000099000111000109000 1..."
>>> "".join([chr(int(doc[i:i+6]))
... for i in range(0,len(doc),6)])
'computing'
```

Text Document Encoding v1 III

- This is the method used for many of the popular non-Unicode character encodings, e.g. **ASCII**
  - 128 characters, based on 7-bit encoding
  - compact, but has the obvious failing that it can only encode a small number of characters
- At other end of extreme, **UTF-32** uses a 32-bit encoding to represent each Unicode code point value directly
  - can encode all Unicode characters, but bloated (documents are much bigger than they need to be)
- How to get the best of both worlds — a compact encoding, but which supports a large character set?

Take 1: Set the "Top Bit"

- **ISO-8859**
  - single-byte encoding built on top of ASCII (7-bit), to include an extra 128 characters, which is enough to represent many orthographies (Latin $\pm$ diacritics, Cyrillic, Thai, Hebrew, ...)
  - as 128 characters isn't enough to cover all these orthographies at once, the standard occurs in 16 variants (ISO-8859-1 to ISO-8859-16) representing different orthographic combinations (e.g. ISO-8859-7 for Latin/Greek)

The Limitations of Single-Byte Encodings

- Single-byte encodings such as ISO-8859-* are great if the alphabet is small, but they aren't able to support "big" orthographies such as Chinese, Japanese and Korean (CJK)
- Also, what if we want to have Greek, Hebrew and Arabic in the same document?

Take 2: Variable-width Encodings

- Encode the code points using a variable number of "code units" of fixed size
- The most popular variable-width Unicode encodings are:
  - **UTF-8**: code unit = 8 bits (1 byte)
  - **UTF-16**: code unit = 16 bits (2 bytes)

UTF-8

- UTF-8 is an 8-bit, variable-width encoding, which is compatible with ASCII

Code range (hex)	UTF-8 (bin)
0x000000–0x00007F	0xxxxxxx
0x000080–0x0007FF	110xxxxx 10xxxxxx
0x000800–0x00FFFF	1110xxxx 10xxxxxx 10xxxxxx
0x010000–0x10FFFF	11110xxx 10xxxxxx 10xxxxxx 10xxxxxx

(i.e. a UTF-8 document containing all code points in the range 0x00–0x7F is identical to an ASCII document)

Features of UTF-8

- Superset of ASCII
- Standard encoding within XML and for HTML5 (and modern operating systems)
- The byte sequence for a given code point never occurs as part of a longer sequence
- Character boundaries are easily locatable

Declaring Character Encodings

- As we have seen, it is not necessarily immediately evident which encoding a given document is encoded in (e.g. consider the ISO-8859-* encodings)
- The primary ways of dealing with this are:
  1. manually specify the "character encoding" in the document, e.g. in the case of HTML::

```
<meta http-equiv="Content-Type"
                content="text/html; charset=iso8859-8" />
```

  2. automatically detect the character encoding, in terms of its compatibility with different encoding standards, user preferences, statistical models, ...

Python and Unicode I

- Mostly in Python3, you don't have to worry about character encodings, as strings are natively in Unicode (unless you want to use other encodings), *except* for file IO:

```
>>> fp = open("unicode.txt")
>>> fp.read()
Traceback (most recent call last):
  File "<pyshell#3>", line 1, in <module>
    print(fp.read())
  File "/usr/lib/python3.4/encodings/ascii.py", ...
    return codecs.ascii_decode(input, self.errors)[0]
UnicodeDecodeError: 'ascii' codec ...
```

Python and Unicode II

- The way around this is to specify the encoding when opening the file, e.g.:

```
>>> fp = open("unicode.txt", encoding="utf-8")
>>> fp.read()
```

Lecture Summary

- What is Unicode, and what problems does it solve?
- What are Unicode code points and code units?
- What is the "ISO-8859" character encoding family, and how does it extend ASCII?
- What are fixed-width vs. variable width encodings?
- What is the relationship between encodings such as `utf-8` and Unicode?