Handout 8: Files and Characters

Reading and writing files

1. Create a plain text file named `foo.txt` containing:

```
Hi there.
This is a test.
```

2. Reading the file

a. "Opening" the file:

```
>>> open('foo.txt')
<open file 'foo.txt', mode 'r' at 0x4023d338>
```

b. A **stream** ("open file" object) is a generator that produces lines.

```
>>> list(open('foo.txt'))
['Hi there.\n', 'This is a test.\n']
```

c. How do we get rid of the newlines? Recall `rstrip` from Handout 7:

```
>>> [line.rstrip('\r\n') for line in open('foo.txt')]
['Hi there.', 'This is a test.']
```

3. The `read` method: special to files, not generators in general.

```
>>> open('foo.txt').read()
'Hi there.\nThis is a test.\n'
```

4. Writing a file

a. Example:

```
>>> f = open('tmp.txt', 'w')
>>> print('Hello world', file=f)
>>> print(42, file=f)
>>> f.close()
```

b. Contents of `tmp.txt` afterwards:

```
Hello world
42
```

5. Using `with`, and tabular files

a. Assume `mobyfd` is FreqDist for *Moby Dick,* `emmafd` for *Emma.*

```
def save_table (fn):
    with open(fn, 'w') as f:
        for w in ['the', 'of', 'whale', 'town']:
            line = '{}\t{}\t{}'.format(w, mobyfd.freq(w), emmafd.freq(w))
            print(line, file=f)
```

b. Reading:

```
def load_table (fn):
    for line in open(fn):
        yield line.rstrip('\r\n').split('\t')
```

Web pages, preprocessing

6. Fetching a web page

```
>>> from urllib.request import urlopen
>>> url = 'http://www.gutenberg.org/files/2554/2554.txt'
>>> bytes = urlopen(url).read()
>>> s = bytes.decode('utf8')
>>> s[:73]
'The Project Gutenberg EBook of Crime and Punishment, by Fyodor Dostoevsky'
```

7. Bytes and strings

a. One byte represents range 0–255. Plain English text (ASCII).

b. Characters from other languages are **encoded** as byte sequences.

```
>>> bytes = b'\xcf\x80\xce\xb5\xcf\x81\xce\xb9'
>>> s = bytes.decode('utf8')
>>> s
'πϵρι'
>>> s.encode('utf8')
b'\xcf\x80\xce\xb5\xcf\x81\xce\xb9'
```

c. The escape sequence "\x80" means the byte with hexadecimal value 80 (decimal value 128).

8. Screen-scraping: use Beautiful Soup

a. Cleans up HTML and creates a parse tree

b. Example: getting the text out

```
>>> from bs4 import BeautifulSoup
>>> url = 'http://google.com/'
>>> bytes = urlopen(url).read()
>>> doc = BeautifulSoup(bytes)
>>> text = doc.get_text()
```

c. That will include a lot of junk. Getting rid of scripts and stylesheets:

```
>>> elts = doc.find_all(lambda x: x.name not in ['script','style'])
>>> [s for e in elts for s in e.children
...  if isinstance(s, str) and not s.isspace()]
...
['Google', 'Search', 'Images', 'Maps', 'Play', ...]
```

9. Tokenization

a. `word_tokenize()` tries to be smart

```
>>> sent1 = 'A.B.C. up $2.50 on 8-30-02 with market-cap $25,000,000.'
>>> word_tokenize(sent1)
['A.B.C.', 'up', '$', '2.50', 'on', '8-30-02', 'with', 'market-cap',
'$', '25,000,000', '.']
```

b. Can be fooled, if the text is odd

```
>>> sent2 = 'The goal-it is true-is 50,10 of which'
>>> word_tokenize(sent2)
['The', 'goal-it', 'is', 'true-is', '50,10', 'of', 'which']
```

c. `wordpunct_tokenize()` is less ambitious, but also less useful

```
>>> wordpunct_tokenize(sent2)
['The', 'goal', '-', 'it', 'is', 'true', '-', 'is', '50', ',', '10',
'of', 'which']
```

Character encodings

10. Bytes, hexadecimal

a. Four bits = 1 hexadecimal digit

0000	0	0100	4	1000	8	1100	*C*
0001	1	0101	5	1001	9	1101	*D*
0010	2	0110	6	1010	*A*	1110	*E*
0011	3	0111	7	1011	*B*	1111	*F*

b. One byte = 8 bits = two hex digits

c. E.g., 71 = 64 + 4 + 2 + 1 = 0100 0111 (binary) = 47 (hex)

d. What is hex 3C in binary? In decimal?

e. What is decimal 90 in binary? In hex?

11. The ASCII character set

a. Codes:

	00		10	20	30	40	50	60	70
0	NUL		c-P	SP	0	@	P	‘	p
1	c-A		c-Q	!	1	A	Q	a	q
2	c-B		c-R	"	2	B	R	b	r
3	c-C	(interrupt)	c-S	#	3	C	S	c	s
4	c-D	(eot)	c-T	$	4	D	T	d	t
5	c-E		c-U	%	5	E	U	e	u
6	c-F		c-V	&	6	F	V	f	v
7	c-G	(bell)	c-W	’	7	G	W	g	w
8	c-H	(backspace)	c-X	(	8	H	X	h	x
9	c-I	(tab)	c-Y	)	9	I	Y	i	y
A	c-J	(newline)	c-Z	*	:	J	Z	j	z
B	c-K	(vtab)	ESC	+	;	K	[	k	{
C	c-L	(formfeed)	FS	,	<	L	\	l	|
D	c-M	(return)	GS	–	=	M	]	m	}
E	c-N		RS	.	>	N	^	n	~
F	c-O		US	/	?	O	_	o	DEL

b. Suppose a file contains the byte sequence

48 65 6C 6C 6F 21 0A

What is it interpreted as plain text? (**Decode** it.)

```
>>> b'\x48\x65\x6c\x6c\x6f\x21\x0a'.decode('ascii')
```

c. **Encode** the text "`The End.\n`" in ASCII.

```
>>> 'The End.\n'.encode('ascii')
b'The End.\n'
>>> list(b'Hi')
[72, 105]
>>> [hex(c) for c in b'Hi']
['0x48', '0x69']
```

d. What is the binary representation for DEL? What can you say about the first bit in any byte in ASCII?

e. How many ASCII **code points** are there?

f. What is the decimal representation for DEL?

12. Latin-1 to Latin-10

a. Different ways of extending ASCII, assigning characters to the "high" code points 80–FF.

b. The Latin-n encodings assigned various accented characters to A0–FF, but in different ways, for different languages.

	Latin-1	Latin-2
A0	NBSP (U+00A0)	NBSP (U+00A0)
A3	£ (U+00A3)	Ł (U+0141)
C1	Á (U+00C1)	Á (U+00C1)
F1	ñ (U+00F1)	ń (U+0144)
FF	ÿ (U+00FF)	˙ (U+0307)

c. What is the decimal value for ÿ? How many code points does Latin-1 have?

d. Code point F1 represents ñ in Latin-1, but it represents ń in Latin-2

```
>>> b'\xf1'.decode('latin1')
'\u00f1'
>>> b'\xf1'.decode('latin2')
'\u0144'
```

e. Windows-1252 ("ANSI") mostly agrees with Latin-1, but assigns characters to the range 80–9F, which are unassigned in Latin-1

13. Unicode

a. 1,114,112 code points. Covers all the world's languages and scripts, and many non-linguistic symbol sets

b. Examples: Cherokee and Canadian native syllabaries, Glagolitic, Gothic, Runic, Linear B, Cuneiform, mathematical symbols, musical notation, ancient Greek and Byzantine musical symbols, Tai Xuan Jing symbols

c. The first 128 code points are identical to ASCII; the first 256 code points are identical to Latin-1.

d. Entering Unicode code points in Python: `'\u0144'`

14. Unicode UTF-16-BE

a. $2^{16} = 65{,}536$ code values. Some characters are represented by pairs of values.

b. Encode "`The End.\n`" in UTF-16-BE

c. ń is U+0144. That is, it is represented by the byte sequence 01 44

d. "BE" for "big-endian." In UTF-16-LE, ń is 44 01.

15. Unicode UTF-8

a. Exactly like ASCII for the first 128 codes. No zero bytes like in UTF-16-BE.

b. Uses sequences of "high" bytes to represent codes above 127. (Not the same as Latin-1.)

c. For example, code point U+0144 is represented as C5 84.

d. U+0801 is represented as E0 A0 81

16. Terminology

a. **(Coded) Character set.** Examples: ASCII, Latin-1, Unicode.

b. **Code point.** Examples: F1 (Latin-2), U+0144 (Unicode).

c. **Character encoding.** Examples: UTF-8, UTF-16-BE, UTF-32-BE

d. **Code value.** 8-bit number in UTF-8, 16-bit number in UTF-16.

e. **Code unit.** Examples: F1 (Latin-2), 01 44 (UTF-16-BE), C5 84 (UTF-8).

f. **Encode**, **decode**

g. **Font**, **glyph**

17. Reading a file with a non-ASCII encoding

a. Finding a file from a corpus: `abspath()`

```
>>> from nltk.corpus import udhr
>>> fn = udhr.abspath('Polish-Latin2')
>>> fn
FileSystemPathPointer('/afs/umich.edu/user/a/b ... /Polish-Latin2')
```

b. Providing encoding to `open`:

```
>>> text = open(fn, encoding='latin2').read()
>>> text[:37]
'POWSZECHNA DEKLARACJA PRAW CZ\u0141OWIEKA\n'
```

c. Get raw bytes and decode

```
>>> bytes = open(fn, encoding='latin2').read()
>>> bytes[:37]
b'POWSZECHNA DEKLARACJA PRAW CZ\xa3OWIEKA\n'
>>> text = bytes.decode('latin2')
>>> text[:37]
'POWSZECHNA DEKLARACJA PRAW CZ\u0141OWIEKA\n'
```

18. Most Unicode characters display properly. But what if you encounter a text that displays as a bunch of `\u` escapes?

a. Write a text file in UTF-8:

```
>>> s = u'\u201c\u6708\u7403\u8f66\u201d'
>>> with open('tmp.txt', 'w', encoding='utf8') as f:
...     print(s, file=f)
...
```

b. Visit `mfile.umich.edu` with a browser, navigate to `tmp.txt`. May need to set character set to Unicode (Firefox: View>Character Encoding).

"月球车"

19. The complete Unicode character set

a. `http://www.unicode.org/Public/UCD/latest/charts/CodeCharts.pdf`

b. 2,186 pages! You can use a PDF viewer and search for particular codes.