Representing text in a computer

Notebook version:
https://github.com/parrt/msds692/blob/master/notes/chars.ipynb

Terence Parr
MSDS program
University of San Francisco

UNIVERSITY OF SAN FRANCISCO

What are characters?

- Data scientists must be able to load data from files and the Internet into memory; often the data is in text form
- Characters are lexical elements that represent words or sounds in a natural language; text is just a bunch of characters
- As with everything else computers represent text using numbers, assigning a unique number to each character
- The way we represent text and memory is often different than in files, so we have to be careful when collecting files from around the world

American characters

- Americans encode the English character set (upper and lower case, numbers, punctuation, and some other characters like newlines and tab) using 7-bit ascii codes: numbers <= 127
- "abc" is represented by three bytes, one byte per character
- It is a very dense encoding, meaning very few bits are wasted
- The encoding is called ASCII and is just a mapping from characters to numbers

Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
32	20	040		Space	64	40	100	@	@	96	60	140	`	`
33	21	041	!	!	65	41	101	A	A	97	61	141	a	a
34	22	042	"	"	66	42	102	B	B	98	62	142	b	b
35	23	043	#	#	67	43	103	C	C	99	63	143	c	c
36	24	044	$	$	68	44	104	D	D	100	64	144	d	d
37	25	045	%	%	69	45	105	E	E	101	65	145	e	e
38	26	046	&	&	70	46	106	F	F	102	66	146	f	f
39	27	047	'	'	71	47	107	G	G	103	67	147	g	g

See http://www.asciitable.com/

A first attempt at non-English characters

- For a very long time, other languages were out of luck
- Other countries started using the remaining 128..255 numeric values to encode characters; e.g. accented letters like ś and ŝ
- This was called the Latin-1 character set
- The problem was that many countries used 201 and to represent different characters; e.g., Russian characters were often mapped to numbers using the KOI8-R set that overlapped with 0..255 used by ASCII and Latin-1

See https://en.wikipedia.org/wiki/ISO/IEC_8859-1

Enter Unicode

- Unicode is a standard that maps characters for just about any human language to numeric values (called *code points*)
- For example, here is a piece of the Bengali code points
- Reading left to right, the first character is 980+0, the second is 980+1, etc...
- Notation U+0981 is hexadecimal, base 16, is used because all possible values within 4 hexadecimal digits (2 bytes)

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

Unicode in Python

- Python strings are a string of Unicode characters (code points)
- Worst-case each character requires two bytes (16 bits) whereas ASCII requires just one byte (8 bits)
- Python 3 does seem to do some optimization, keeping strings as 1-byte-per-char as long as possible, until we introduce a non-ASCII character
- We can verify string size with the **getsizeof** function

Checking string memory requirements

```
   from sys import getsizeof
49 print(getsizeof(''))   # 49 bytes of overhead for string object
50 print(getsizeof('a'))
51 print(getsizeof('ab'))
52 print(getsizeof('abc'))
76 print(getsizeof('Ω'))  # add non-ASCII char & overhead goes up
78 print(getsizeof('ΩΩ')) # each unicode char costs 2 bytes
80 print(getsizeof('ΩΩΩ'))
```

Unicode character names

```
import unicodedata
print(unicodedata.name(chr(9999)))
print(unicodedata.name(chr(9991)))
# sequence "\N" means named entity
print("\N{GREEK CAPITAL LETTER OMEGA}")
print("\N{PENCIL}")
print("\N{TAPE DRIVE}")
```

PENCIL
TAPE DRIVE
Ω
✏️
✇

Converting character codes to chars

- The **chr**() function converts an integer to a character

Output	Code
d	`print(chr(100))`
ፋ	`print(chr(4939))`
ô 'ô'	`print(chr(244), repr(chr(244)))`

- You will see notation **\xFF**, which means **FF** in hexadecimal (all bits on) or 255 in decimal
- A byte takes at most 2 hexadecimal digits, which is why we tend to use hexadecimal
- **\uABCD** notation is used to express 2-byte (16 bit) Unicode chars

```
[4]: '\u00ab'
```

```
[4]: '«'
```

Chars to integer code points

```
ord('Ω')
```

```
937
```

```
chr(ord('Ω'))
```

```
'Ω'
```

```
[ord(c) for c in 'hiፋ']
```

```
[104, 105, 4939]
```

```
[hex(ord(c)) for c in 'hiፋ'] # show ord values in hex
```

```
['0x68', '0x69', '0x134b']
```

Exercise

- Repeat for yourself the notebook cells above starting with the **getsizeof** stuff to get used to playing with non-English characters, converting them to and from their code points (ordinal values)
- You can cut/paste some stuff from the notebook version of this lecture: https://github.com/parrt/msds692/blob/master/notes/chars.ipynb

Text file encoding

Now, let's make a distinction between strings in memory and text files stored on the disk

UNIVERSITY OF SAN FRANCISCO

ASCII text files

- Storing Python strings with ASCII chars, codes that fit into 8 bits (1 byte), into a file is easy
- The sequence of character codes is stored in the file one byte per character
- That is a very dense encoding
- Using a compression algorithm we could make the file smaller but it would no longer be a text file

Storing UNICODE into text files

- For 16-bit Unicode, we could store each character as to bytes in the file, but it wastes a lot of space; English characters would require double the space, for example!
- Instead of blindly storing two bytes per character, we should optimize for the case where characters fit into one byte
- We use such an encoding called UTF-8: *Unicode Transformation Format* that is optimized for ASCII char

UTF-8 degenerates to ASCII for ASCII chars

UTF-8

- The details don't matter for us, but the table below summarizes how many bytes are required for each Unicode value
- It's just an efficient way to store Unicode characters in a file; remember that in a string in memory, we should think of characters is always taking exactly 2 bytes (16 bits)

1st Byte	2nd Byte	3rd Byte	4th Byte	Number of Free Bits	Maximum Expressible Unicode Value
0xxxxxxx				7	007F hex (127)
110xxxxx	10xxxxxx			(5+6)=11	07FF hex (2047)
1110xxxx	10xxxxxx	10xxxxxx		(4+6+6)=16	FFFF hex (65535)
11110xxx	10xxxxxx	10xxxxxx	10xxxxxx	(3+6+6+6)=21	10FFFF hex (1,114,111)

Other file encodings

- There are non-UTF-8 encodings of strings for files
- For example, on a Japanese machine, the encoding might be *euc-jp*, which is optimized for the Japanese character set. (Wikipedia says "*EUC-JP is a variable-width encoding used to represent the elements of three Japanese character set standards,...*")
- **Bottom line:** If you are reading text from a file, you must know the encoding in order to communicate; A file from Japan might not have the same encoding as a file created locally on your US machine, even with identical text content

Saving text into files

Writing ASCII text to a file

```
with open("/tmp/ascii.txt", mode="w") as f:
    f.write("ID 345\n")
```

```
varmint:/tmp $ python ascii.py
varmint:/tmp $ od -c -t dC /tmp/ascii.txt # chars in decimal
0000000    I   D        3   4   5  \n
          73  68  32   51  52  53  10
0000007
varmint:/tmp $ od -c -t xC /tmp/ascii.txt # chars in hex
0000000    I   D        3   4   5  \n
          49  44  20   33  34  35  0a
0000007
```

Please note that 345 is a sequence of three characters not the binary value 345

Writing non-ASCII char to a file

```
# Write a UTF-8-encoded text file
with open('/tmp/utf8.txt', encoding='utf-8', mode='w') as f:
    f.write('Pencil: \N{PENCIL}, Euro: \u20ac\n')
    # or use actual character:
    # f.write('Pencil: ✏️, Euro: €\n')
```

```
varmint:/tmp $ od -c -t xC /tmp/utf8.txt
0000000    P   e   n   c   i   l   :       ✏  **  **   ,       E   u   r
          50  65  6e  63  69  6c  3a  20  e2  9c  8f  2c  20  45  75  72
0000020    o   :       €  **  **  \n
          6f  3a  20  e2  82  ac  0a
0000027
```

The ** mean "Hi, I'm a byte that is part of the preceding character shown"

Reading Unicode encoded as UTF-8

- The encoding used to read must match encoding used to write

```
with open('/tmp/utf8.txt', encoding='utf-8', mode='r') as f:
    s = f.read()
print(s)
```

Pencil: ✏️, Euro: €

- If you use the wrong encoding you get the wrong strings:

```
with open('/tmp/utf8.txt', encoding='latin-1', mode='r') as f:
…
```

Pencil: â□□, Euro: â□¬

Exercise

- Test out those two simple Python programs to make sure you can **write** and **read** Unicode characters to and from files. But change the string so your code saves two characters: VICTORY HAND followed by HEAVY CHECK MARK or some other fun characters
- Use the **od** command to dump the characters in the file
- You can cut/paste some stuff from the notebook version of this lecture: https://github.com/parrt/msds692/blob/master/notes/chars.ipynb

Language within a text file

- Besides knowing that a file is text and how it is encoded, we also need to know the format or language followed by the characters; there are many different formats (syntax):
  - comma-separate values (CSV)
  - XML
  - JSON
  - HTML
  - Natural language text, such as an email message or tweet
  - Python, JavaScript, Java, C++, any programming language
- Examples of non-text-based formats: mp3, png, jpg, mpg, ...

Some formats have a lot of overhead

- Take a look at the sizes to represent the same data in four different formats for your pipeline project (AAPL stock prices):

```
$ ls -l
total 9728
-rw-r--r--  1 parrt  wheel   583817 Aug 25 16:21 AAPL.csv
-rw-r--r--  1 parrt  wheel  1177603 Aug 25 16:21 AAPL.html
-rw-r--r--  1 parrt  wheel  1438395 Aug 25 16:21 AAPL.json
-rw-r--r--  1 parrt  wheel  1771234 Aug 25 16:21 AAPL.xml
```

- CSV is the least flexible but the most dense text format