Understanding Data

Number Systems and Text Encoding

Number Systems

Base 10 (Decimal numbers)

Why base 10?

We *happened* to use the current counting system, because we happened to have ten fingers.

Base 10(Decimal) uses 10 digits {0,1,2,3…,9}.

Base 2(Binary) uses 2 digits {0,1}.

Base 8(Octal) uses 8 digits {0,1,2,3,4,5,6,7}

Base 10 (Decimal numbers)

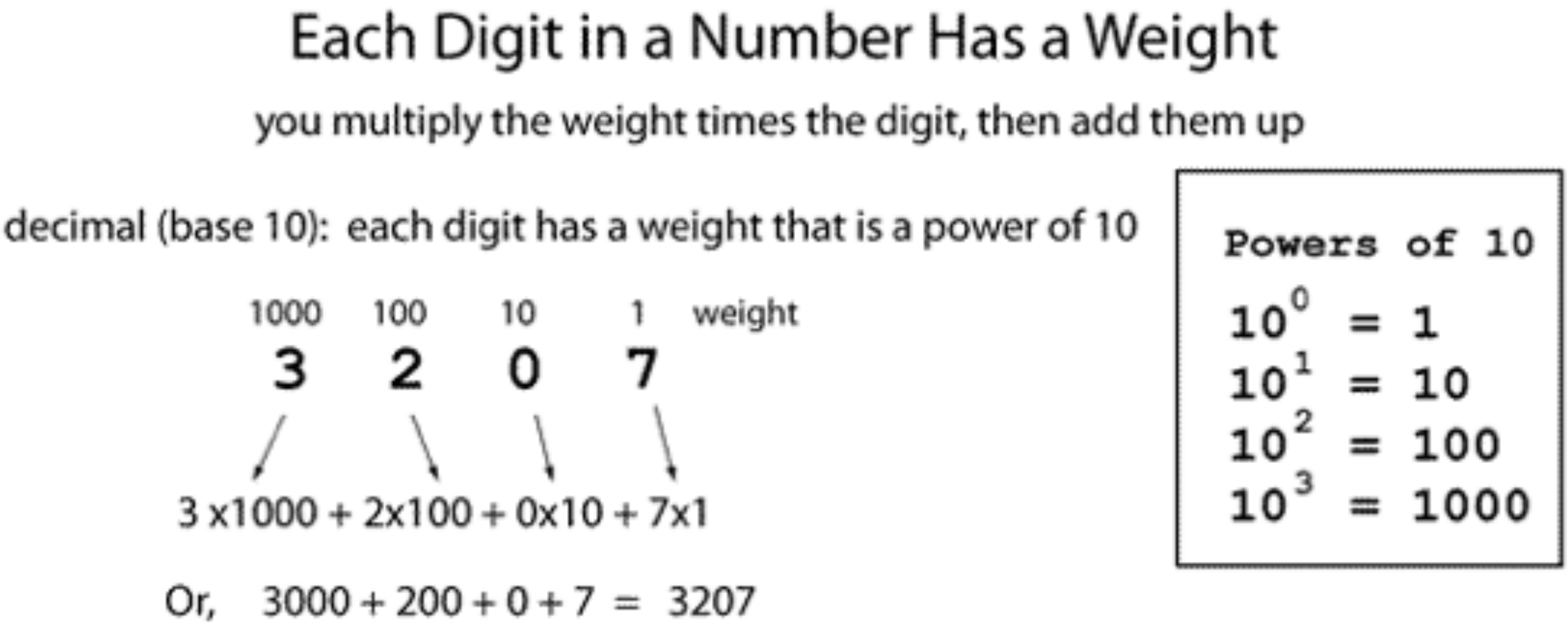
Each digit has a weight corresponding to a power of 10.
Multiply each digit with its weight and then sum.

What does 157 mean?

$157 = 1 \times 100 + 5 \times 10 + 7 \times 1$

$= 1 \times 10^2 + 5 \times 10^1 + 7 \times 10^0$

Base 10 (Decimal)



Each Digit in a Number Has a Weight

you multiply the weight times the digit, then add them up

decimal (base 10): each digit has a weight that is a power of 10

1000	100	10	1	weight
3	2	0	7	

3 x1000 + 2x100 + 0x10 + 7x1

Or, 3000 + 200 + 0 + 7 = 3207

Powers of 10

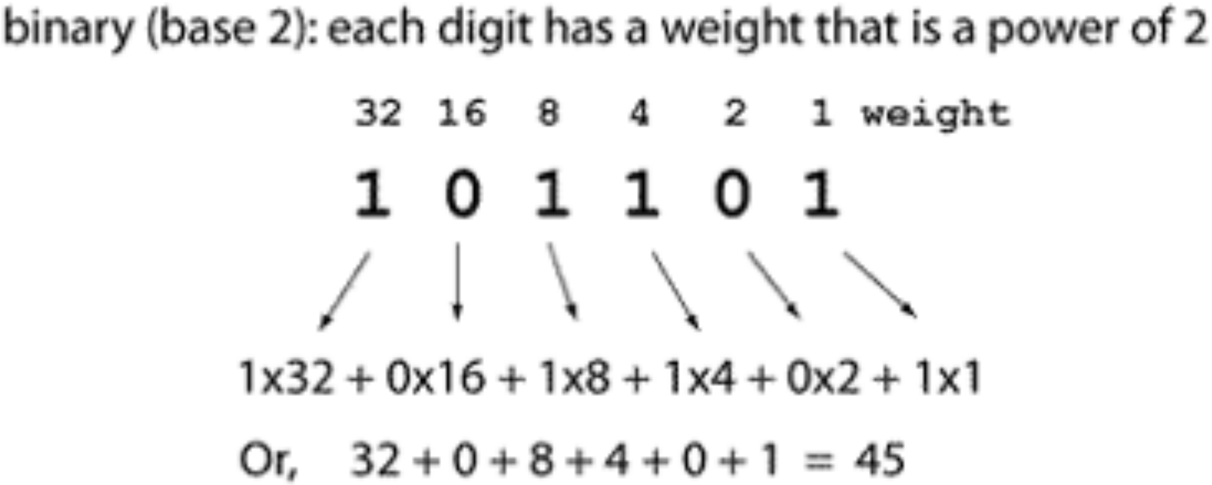
$10^0 = 1$

$10^1 = 10$

$10^2 = 100$

$10^3 = 1000$

Base 2 (Binary)



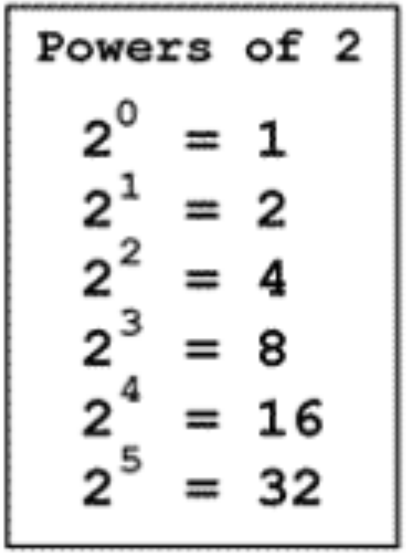
binary (base 2): each digit has a weight that is a power of 2

```
32  16   8    4    2    1  weight

1   0    1    1    0    1
```

1x32 + 0x16 + 1x8 + 1x4 + 0x2 + 1x1

Or, 32 + 0 + 8 + 4 + 0 + 1 = 45



Powers of 2

$2^0 = 1$

$2^1 = 2$

$2^2 = 4$

$2^3 = 8$

$2^4 = 16$

$2^5 = 32$

Base 2 (Binary)

Base 2

$1011 = 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0$

$1011 = 1 \times 8 + 0 \times 4 + 1 \times 2 + 1 \times 1 = 11$ in Base 10

Bits and Bytes

1 **bit** is a single bit of information, a 1 or 0(Only two possible values)

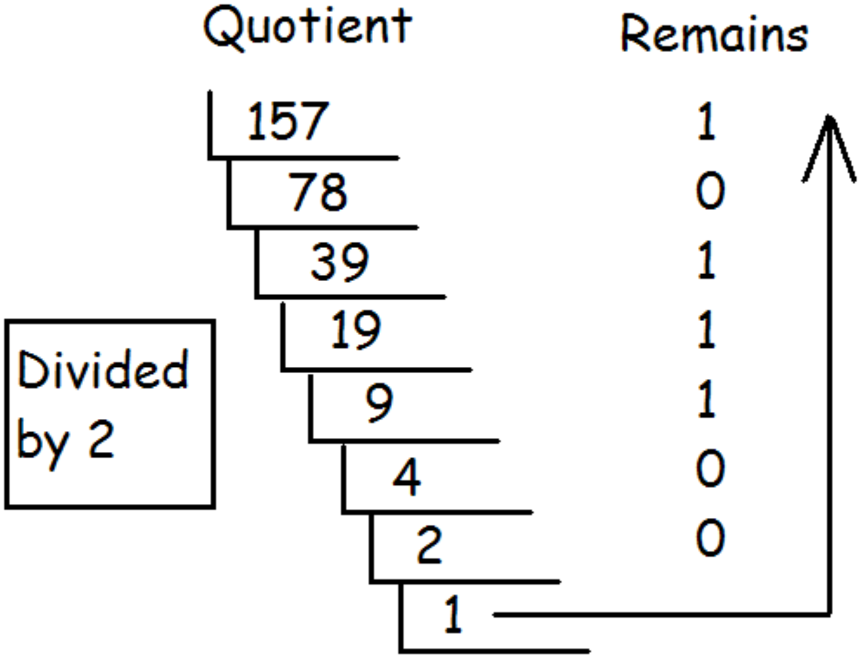
1 **byte** is 8 bits, an 8 bit word

- 256 possible values from 0-255 **base 10**
- or 00000000 to 11111111 **base 2**

For example, 10100110 is a single byte

Decimal to Binary

Repeatedly modulo 2 and the divide by 2. Stop at 0. The remainder list read top to bottom are the digits from left to right.



Divided by 2

Quotient	Remains
157	1
78	0
39	1
19	1
9	1
4	0
2	0
1	

Answer: 10011101

Adding Binary

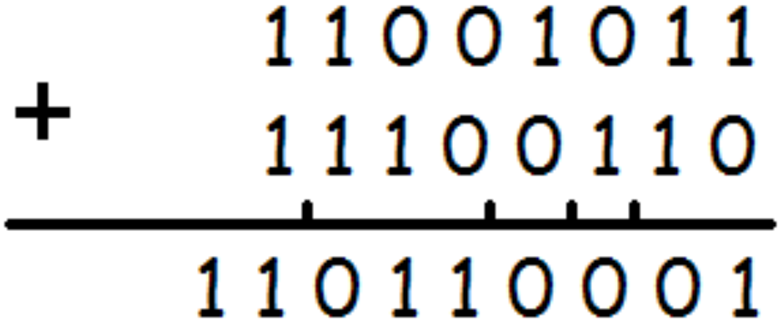
Add from right to left as usual. Follow the three rules below and remember to carry.

$$0 + 0 = 0$$

$$1 + 0 = 1$$

$$1 + 1 = 10$$

Example:



```
     11001011
+    11100110
  -----------
    110110001
```

People

There are 10 types of people in the world; those who understand binary and those who don’t.

There are 10 types of people in the world; those who understand binary and those who have friends.

Hexadecimal

Binary code is too long in representation. **Hexadecimal** is much shorter.
Hexadecimal uses 16 digits

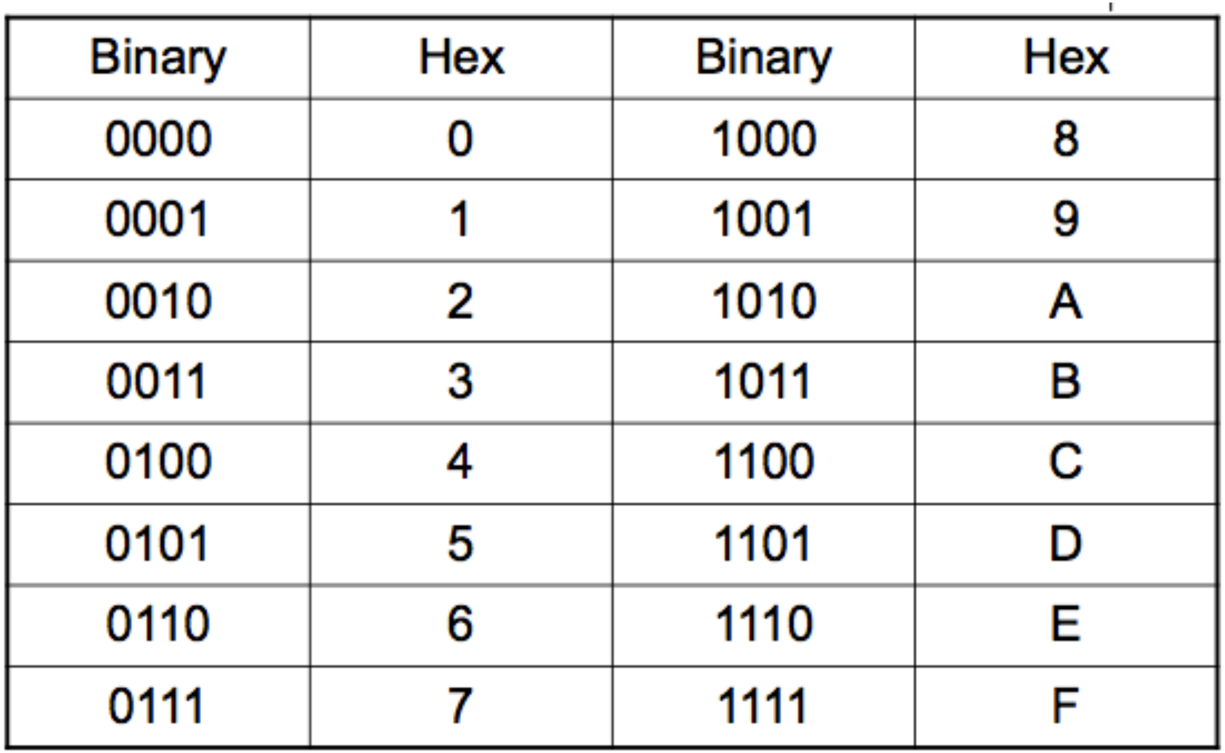
Problem: we are short of numbers.

A-10 B-11 C-12 D-13 E-14 F-15

Hexadecimal digits: {0,1,2,3,4…,9,A,B,C,D,E,F}

Converting a binary number to a Hex number is relatively easy. Every 4 bit can convert to a Hex.

Hexadecimal



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

Binary to Hex

$$\underbrace{1\,0\,1\,1}\,\underbrace{1\,1\,1\,1}\,\underbrace{0\,0\,1\,0}\,\underbrace{1\,1\,0\,0}$$

Dec	11	15	2	12
Hex	B	F	2	C

Result BF2C

Counting

Let’s count in Base 10.

0,1,2,3,4,5,6,7,8,9,__,___,…,18,19,__,__.

Answer: 10,11…..,20,21. Good job!

Let’s count in Base 5.

0,1,2,3,4,__,__,__,__,__,___.

Answer:10,11,12,13,14,20.

Let’s keep going!

40,41,42,43,44,___,___,___.

Answer:100,101,102.

Quick Quiz

What's the answer in base 8?

$$6_{10} + 3_{10} = 11_8$$

What's the answer in base 5?

$$2_{10} + 6_{10} = 13_5$$

What's the answer in base 12?

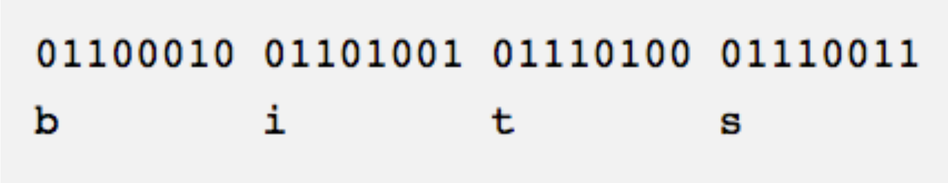
$$9_{10} + 15_{10} = 20_{12}$$

Character Encoding

Encoding

A computer cannot store “letters” or “pictures”. It can only work with bits(0 or 1).

To represent anything other than bits, we need to rules that will allow us to convert a sequence of bits into letters or pictures. This set of rules is called an **encoding scheme**, or **encoding** for short. For example,



```
01100010 01101001 01110100 01110011
b        i        t        s
```

ASCII

ASCII(American Standard Code for Information Exchange) is an encoding scheme that specifies the mapping between bits and characters. There are 128 characters in the scheme. There are 95 readable characters and 33 values for nonprintable characters like space, tab, and backspace and so on.

The readable characters include a-z, A-Z, 0-9 and punctuation.

ASCII uses all possible combinations of 7 bits(0000000 to 1111111, 2^7=128).

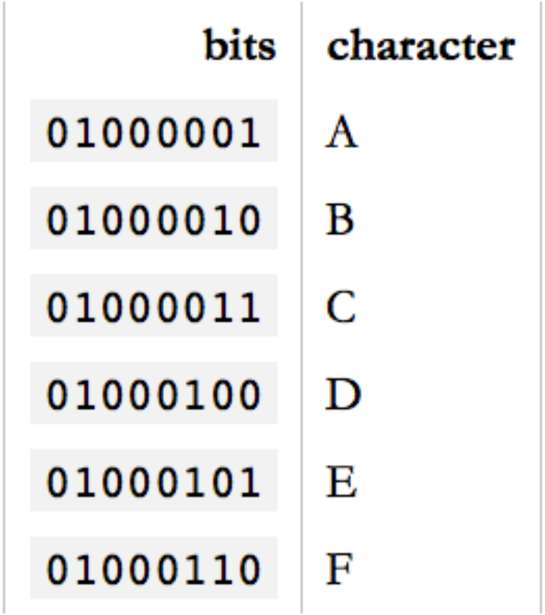
ASCII Sample

In ASCII, 65 represents A, 66 represents B and so on. The numbers are called code points.

To **encode** something in ASCII, follow the table from right to left, substituting letters for bits.

To **decode** a string of bits into human readable characters, follow the table from left to right, substituting bits for letters.

encode



bits	character
01000001	A
01000010	B
01000011	C
01000100	D
01000101	E
01000110	F

decode

Other letters?

95 characters can covers English but what about French? Or how do you represent Swedish letters é, ß, ü, ä, ö or å in ASCII?

"But look at it," the Europeans said, "in a common computer with 8 bits to the byte, ASCII is wasting an entire bit which is always set to 0! We can use that bit to squeeze a whole 'nother 128 values into that table!" And so they did.

The extended ASCII table has 256 values. All 8 bits were used.

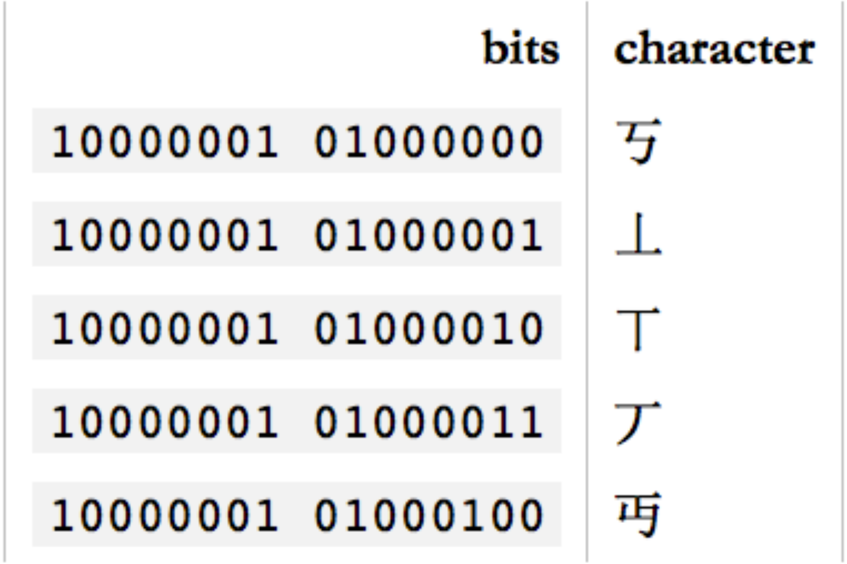
Other letters

But even so, there are more than 128 ways to stroke, slice, slash and dot a vowel. Not all variations of letters and squiggles used in all European languages can be represented in the same table with a maximum of 256 values.

Not to mention Chinese. Or Japanese. Both contains thousands of characters. There are many encodings that try to address this. For example, BIG-5 covers all traditional Chinese characters.

GB18030 Encoding

And GB18030 covers both traditional and simplified Chinese as well as a large part of the latin characters.



bits	character
10000001 01000000	丂
10000001 01000001	丄
10000001 01000010	丅
10000001 01000011	丆
10000001 01000100	丏

GB18030 uses two bytes (16 bits) to encode. This can encode 65,536 distinct values.

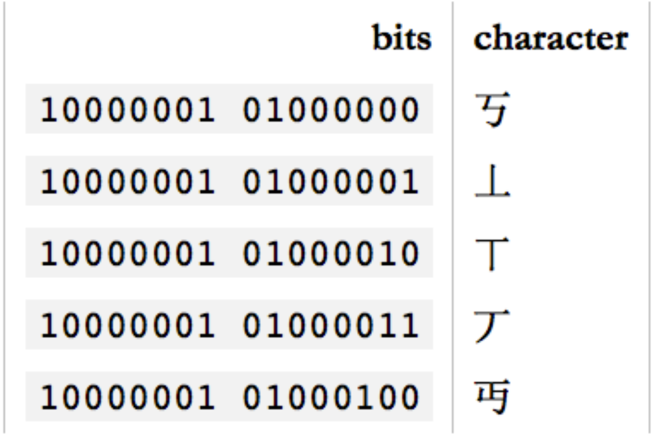
But in the end, GB18030 is yet another specialized encoding among many.

Encoding/Decoding Documents

In an editor such as Word document, you can specify the language encoding.

Special software can help you type special characters.

When you save the document, the editor saves the character as 0's And 1's as specified by the encoding.



bits	character
10000001 01000000	丂
10000001 01000001	丄
10000001 01000010	丅
10000001 01000011	丆
10000001 01000100	丏

When you reopen the document, the encoding setting will allow the editor to decode the 0's and 1's back to correct characters.

Unicode

There are many encoding schemes for all sorts of applications. We need a standardized encoding.

Finally, in 1991, **Unicode** was created to unify all encoding standards. It defines a table of 1,114,112 code points that can be used for all sorts of letters and symbols. That's plenty to encode all European, Middle-Eastern, Far-Eastern, Southern, Northern, Western, pre-historian and future characters mankind knows about.

Unicode

Using Unicode, you can write a document containing virtually any language using any character you can type into a computer. This was either impossible or very very hard to get right before Unicode came along. There's even an unofficial section of Klingon in Unicode.

Unicode

So, how many bits does Unicode use to encode all these characters? *None*. Because Unicode is not an encoding.

Unicode first and foremost defines a table of ***code points*** for characters. That's a fancy way of saying "*65 stands for A, 66 stands for B and 9,731 stands for ☃*".

UTF-32

To represent 1,114,112 different values, two bytes aren't enough. Three bytes are, but three bytes are often awkward to work with, so four bytes would be the comfortable minimum.

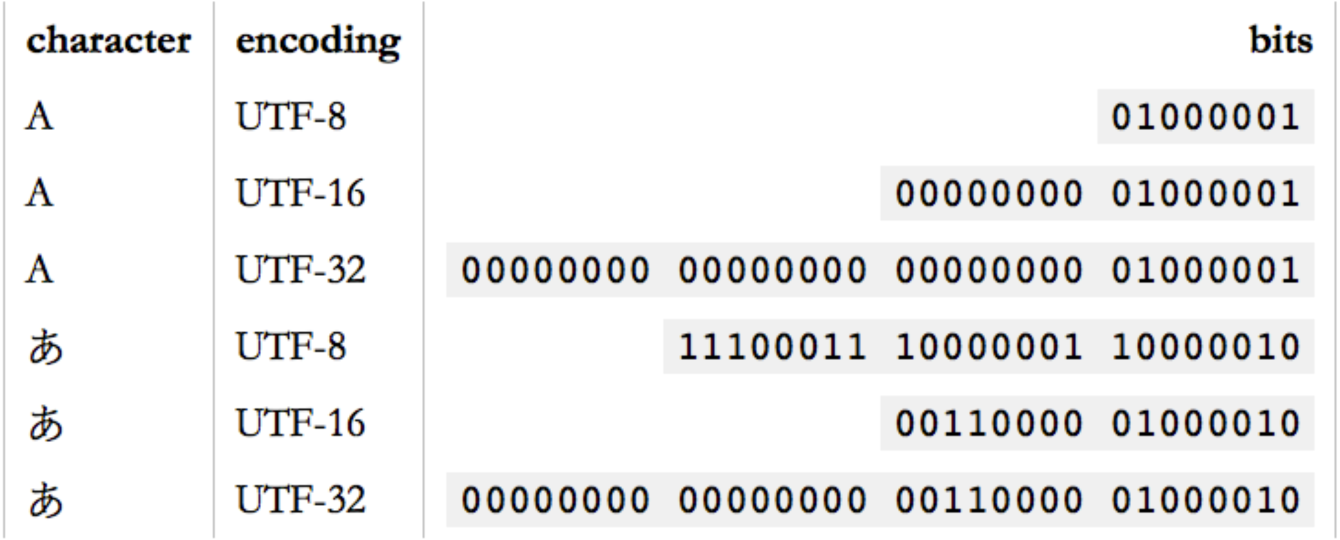
UTF-32(Unicode Transformation Format) is such an encoding that encodes all Unicode code points using 32 bits. That is, four bytes per character.

UTF-32

UTF-32 very simple, but often wastes a lot of space. For example, if A is always encoded as 00000000 00000000 00000000 01000001 and B as 00000000 00000000 00000000 01000010 and so on, documents would bloat to 4x its necessary size.

UTF-16 and **UTF-8** are *variable-length encodings*. If a character can be represented using a single byte (because its code point is a very small number), UTF-8 will encode it with a single byte. If it requires two bytes, it will use two bytes and so on. UTF-16 is in the middle, using at least two bytes, growing to up to four bytes as necessary.

Unicode



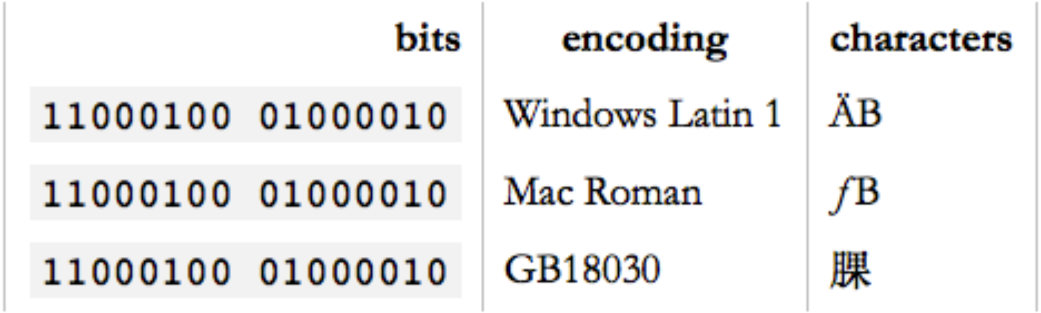
character	encoding	bits
A	UTF-8	01000001
A	UTF-16	00000000 01000001
A	UTF-32	00000000 00000000 00000000 01000001
あ	UTF-8	11100011 10000001 10000010
あ	UTF-16	00110000 01000010
あ	UTF-32	00000000 00000000 00110000 01000010

Unicode is a large table mapping characters to numbers and the different UTF encodings specify how these numbers are encoded as bits.

UTF-32

Any character can be encoded in many different bit sequences and any particular bit sequence can represent many different characters, depending on which encoding is used to read or write them.

The reason is simply because different encodings use different numbers of bits per characters and different values to represent different characters.



bits	encoding	characters
11000100 01000010	Windows Latin 1	ÄB
11000100 01000010	Mac Roman	ƒB
11000100 01000010	GB18030	腂

Garbled Text?

You open a text file and it looks like:

ÉGÉìÉRÅ[ÉfÉBÉìÉOÇÕìÔÇµÇ≠Ç»Ç¢

Reason: Your text editor, browser, word processor or whatever else that's trying to read the document is assuming the wrong encoding.

In Safari or Firefox, you can change the text encoding. (Chrome only supports this through an extension.) For Safari, under “View”, select “Text Encoding” and “Japanese Shift JS” to see the correct text:

エンコーディングは難しくない

UTF-8

UTF-8 is the standard encoding for all email and webpages(HTML5).

The ingenious thing about UTF-8 is that it's binary compatible with ASCII, which is the de-facto baseline for all encodings.

All characters available in the ASCII encoding only take up a single byte in UTF-8 and they're the exact same bytes as are used in ASCII. In other words, ASCII maps 1:1 unto UTF-8. Any character not in ASCII takes up two or more bytes in UTF-8.

We will explore more on text encoding with Python in the next lecture.

Homework

1) Read and reread these lecture notes.
2) Do the problem set.

References

1) What Every Programmer Absolutely, Positively Needs To Know About Encodings And Character Sets To Work With Text, by David Zentgraf. http://kunststube.net/encoding/

2) Part of this lecture is taken from a lecture from an OpenCourseWare course below.

Computer Science E-1 at Harvard Extension School

Understanding Computers and the Internet

by Tommy MacWilliam.