Representing Data

Digitizing our world

Everything in the computer is stored as a number. This includes numbers of course as well as letters, audio files, movie files, etc.

Unary

"*One if by Land Two if by Sea*" –Paul Revere

A single symbol or digit: 1. It's the way we count with tick marks in groups of 5. |, ||, |||, ||||, 𝍸

Unary digits			Value
			0
		1	1
	1	1	2
1	1	1	3

Binary

Numbers are encoded in the machine using binary, 0 or 1 which corresponds to usually 0 and +5 Volts in the hardware. The fundamental element within a computer is a switch that can be either on or off just like a lightbulb. If I have one lightbulb, I can have two states: on or off, 0 or 1, lo or hi, whatever you want to call it.

Q. If I have two lightbulbs, how many states can I have? 4

Binary digits		Value
0	0	0
0	1	1
1	0	2
1	1	3

What about three? $2^3 = 8$:

Binary digits			Value	Count
0	0	0	0	1
0	0	1	1	2
0	1	0	2	3
0	1	1	3	4
1	0	0	4	5
1	0	1	5	6
1	1	0	6	7
1	1	1	7	8

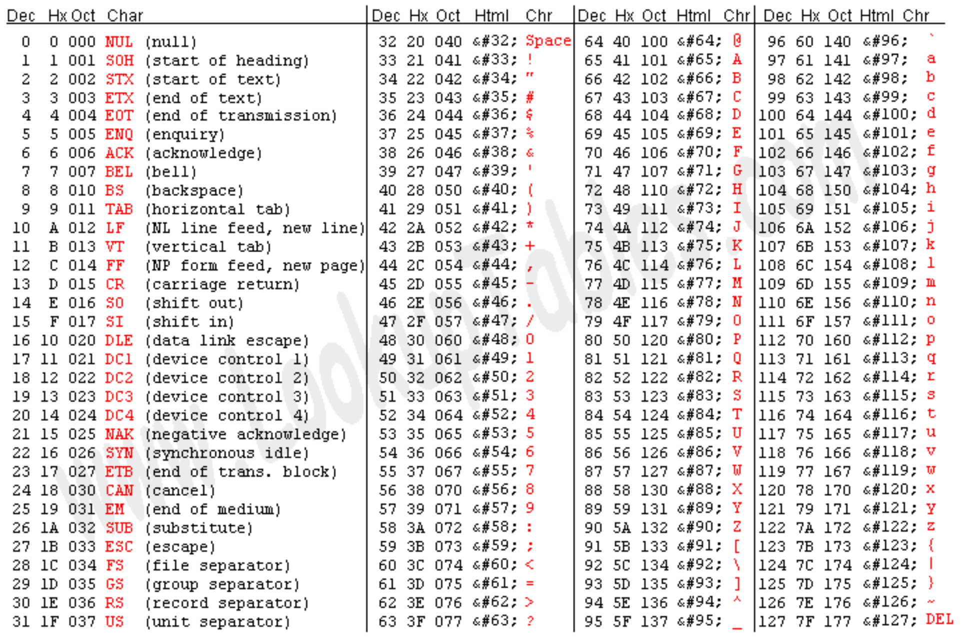
These are two and three bit numbers, which indicates their maximum value.

A **byte** is 8 bits. $2^8 = 256$ states or values 0..255. A **word** is typically the size of the microprocessor's registers. These days that will be 64 bits.

Binary numbers get long pretty quickly: `1010011010` binary is 666 decimal.

Characters

Everything is a number, so how to represent letters? (For now we'll stick with the American character set). We assign a unique number to each letter:



Dec	Hx	Oct	Char		Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL	(null)	32	20	040	&#32;	Space	64	40	100	&#64;	@	96	60	140	&#96;	`
1	1	001	SOH	(start of heading)	33	21	041	&#33;	!	65	41	101	&#65;	A	97	61	141	&#97;	a
2	2	002	STX	(start of text)	34	22	042	&#34;	"	66	42	102	&#66;	B	98	62	142	&#98;	b
3	3	003	ETX	(end of text)	35	23	043	&#35;	#	67	43	103	&#67;	C	99	63	143	&#99;	c
4	4	004	EOT	(end of transmission)	36	24	044	&#36;	$	68	44	104	&#68;	D	100	64	144	&#100;	d
5	5	005	ENQ	(enquiry)	37	25	045	&#37;	%	69	45	105	&#69;	E	101	65	145	&#101;	e
6	6	006	ACK	(acknowledge)	38	26	046	&#38;	&	70	46	106	&#70;	F	102	66	146	&#102;	f
7	7	007	BEL	(bell)	39	27	047	&#39;	'	71	47	107	&#71;	G	103	67	147	&#103;	g
8	8	010	BS	(backspace)	40	28	050	&#40;	(	72	48	110	&#72;	H	104	68	150	&#104;	h
9	9	011	TAB	(horizontal tab)	41	29	051	&#41;	)	73	49	111	&#73;	I	105	69	151	&#105;	i
10	A	012	LF	(NL line feed, new line)	42	2A	052	&#42;	*	74	4A	112	&#74;	J	106	6A	152	&#106;	j
11	B	013	VT	(vertical tab)	43	2B	053	&#43;	+	75	4B	113	&#75;	K	107	6B	153	&#107;	k
12	C	014	FF	(NP form feed, new page)	44	2C	054	&#44;	,	76	4C	114	&#76;	L	108	6C	154	&#108;	l
13	D	015	CR	(carriage return)	45	2D	055	&#45;	-	77	4D	115	&#77;	M	109	6D	155	&#109;	m
14	E	016	SO	(shift out)	46	2E	056	&#46;	.	78	4E	116	&#78;	N	110	6E	156	&#110;	n
15	F	017	SI	(shift in)	47	2F	057	&#47;	/	79	4F	117	&#79;	O	111	6F	157	&#111;	o
16	10	020	DLE	(data link escape)	48	30	060	&#48;	0	80	50	120	&#80;	P	112	70	160	&#112;	p
17	11	021	DC1	(device control 1)	49	31	061	&#49;	1	81	51	121	&#81;	Q	113	71	161	&#113;	q
18	12	022	DC2	(device control 2)	50	32	062	2	2	82	52	122	&#82;	R	114	72	162	&#114;	r
19	13	023	DC3	(device control 3)	51	33	063	&#51;	3	83	53	123	&#83;	S	115	73	163	&#115;	s
20	14	024	DC4	(device control 4)	52	34	064	&#52;	4	84	54	124	&#84;	T	116	74	164	&#116;	t
21	15	025	NAK	(negative acknowledge)	53	35	065	&#53;	5	85	55	125	&#85;	U	117	75	165	&#117;	u
22	16	026	SYN	(synchronous idle)	54	36	066	&#54;	6	86	56	126	&#86;	V	118	76	166	&#118;	v
23	17	027	ETB	(end of trans. block)	55	37	067	&#55;	7	87	57	127	&#87;	W	119	77	167	&#119;	w
24	18	030	CAN	(cancel)	56	38	070	&#56;	8	88	58	130	&#88;	X	120	78	170	&#120;	x
25	19	031	EM	(end of medium)	57	39	071	&#57;	9	89	59	131	&#89;	Y	121	79	171	&#121;	y
26	1A	032	SUB	(substitute)	58	3A	072	&#58;	:	90	5A	132	&#90;	Z	122	7A	172	&#122;	z
27	1B	033	ESC	(escape)	59	3B	073	&#59;	;	91	5B	133	&#91;	[	123	7B	173	&#123;	{
28	1C	034	FS	(file separator)	60	3C	074	&#60;	<	92	5C	134	&#92;	\	124	7C	174	&#124;	|
29	1D	035	GS	(group separator)	61	3D	075	&#61;	=	93	5D	135	&#93;	]	125	7D	175	&#125;	}
30	1E	036	RS	(record separator)	62	3E	076	&#62;	>	94	5E	136	&#94;	^	126	7E	176	&#126;	~
31	1F	037	US	(unit separator)	63	3F	077	&#63;	?	95	5F	137	&#95;	_	127	7F	177	&#127;	DEL

Source: www.LookupTables.com

You can think of this is kind of an encryption. Instead of saying "Hi" you might say "73 105."

There are multiple standards but the clear winner is ASCII. In the old days there was also EBCDIC.

As we will see shortly, a phrase or sentence or word is just a sequence of characters hence a sequence of numbers stored in the machine.

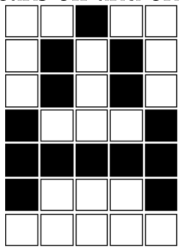
```
$ cat sentence.txt
As we will see shortly, a phrase or sentence or word is just a sequence
of characters hence a sequence of numbers stored in the machine.
$ od -b sentence.txt
0000000   101 163 040 167 145 040 167 151 154 154 040 163 145 145 040 163
0000020   150 157 162 164 154 171 054 040 141 040 160 150 162 141 163 145
0000040   040 157 162 040 163 145 156 164 145 156 143 145 040 157 162 040
0000060   167 157 162 144 040 151 163 040 152 165 163 164 040 141 040 163
0000100   145 161 165 145 156 143 145 012 157 146 040 143 150 141 162 141
0000120   143 164 145 162 163 040 150 145 156 143 145 040 141 040 163 145
0000140   161 165 145 156 143 145 040 157 146 040 156 165 155 142 145 162
0000160   163 040 163 164 157 162 145 144 040 151 156 040 164 150 145 040
0000200   155 141 143 150 151 156 145 056 012
0000211
$ od -c -b sentence.txt
0000000   101 163 040 167 145 040 167 151 154 154 040 163 145 145 040 163
            A   s       w   e       w   i   l   l       s   e   e       s
```

```
0000020   150 157 162 164 154 171 054 040 141 040 160 150 162 141 163 145
            h   o   r   t   l   y   ,       a       p   h   r   a   s   e
0000040   040 157 162 040 163 145 156 164 145 156 143 145 040 157 162 040
                o   r       s   e   n   t   e   n   c   e       o   r
0000060   167 157 162 144 040 151 163 040 152 165 163 164 040 141 040 163
            w   o   r   d       i   s       j   u   s   t       a       s
0000100   145 161 165 145 156 143 145 012 157 146 040 143 150 141 162 141
            e   q   u   e   n   c   e  \n   o   f       c   h   a   r   a
0000120   143 164 145 162 163 040 150 145 156 143 145 040 141 040 163 145
            c   t   e   r   s       h   e   n   c   e       a       s   e
0000140   161 165 145 156 143 145 040 157 146 040 156 165 155 142 145 162
            q   u   e   n   c   e       o   f       n   u   m   b   e   r
0000160   163 040 163 164 157 162 145 144 040 151 156 040 164 150 145 040
            s       s   t   o   r   e   d       i   n       t   h   e
0000200   155 141 143 150 151 156 145 056 012
            m   a   c   h   i   n   e   .  \n
0000211
```

Images

Images are stored as numbers also.

For black and white images, we can use a single bit to represent a black-and-white pixel where zero means off and one means on:



The bit sequence is:

```
00100 01010 01010 10001 11111 10001 00000
```

If you stack vertically, you can see the image sort of:

```
00100
01010
01010
10001
11111
10001
00000
```

Each pixel on the screen is typically represented by three numbers, though: (red, green, blue) RGB values. For example:

white: 255 255 255 (yes they are one byte each)

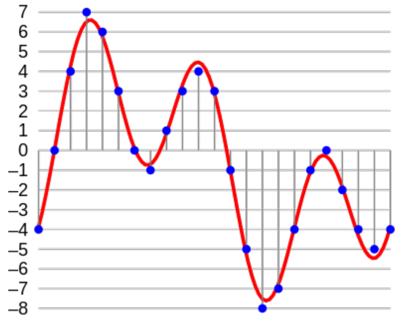
black: 0 0 0

blue: 0 0 255 (blue is saturated)

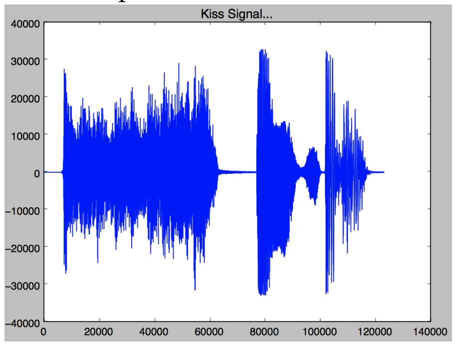
Yellow is a mix of red and green: 255 255 0

Audio

Audio files that you listen to are also represented as just a sequence of numbers where each number represents the amplitude of the signal at discrete locations and time. From Wikipedia page on Digital audio:



Here is a partial waveform of Prince's "Kiss" song:



I loaded via a sample program in `notes/code/plotaiff.py`:

```
$ cd notes/code
$ python plotaiff.py ../../data/Kiss.aiff
$ open ../../data/Kiss.aiff # play the audio
```

(*BTW, that "*`..`*" means jump up a directory*)

Q. What to those values represent?

Q. What happens if you scale each value?

```
$ python scaleaiff.py ../data/Kiss.aiff /tmp/scaled-kiss.aiff
$ open /tmp/scaled-kiss.aiff
```

Sample code: plotaiff.py, scaleaiff.py.

Entropy

It could be a good time to mention the term entropy, which is a measure of the chaos or disorder of a model or system. In the real world, systems always tend towards increased entropy. For example, the state of my kitchen approaches maximum entropy two weeks after the maid has cleaned up. You will see entropy again when you look at algorithms to construct random forests. Entropy in information theory describes how much information is in a signal.

Q. Does it take more bits or fewer bits to store random noise compared to, for example, a pure tone at a particular frequency?

If you need to store a random variable that can take n values with equal probability, you need $log_2(n)$ bits to represent that variable/number. On the other hand, if we know for sure that the random variable is always, say, 9345 or 0 then it only takes a single bit to represent that random variable (9345 is present or not).

Python's atomic elements

There are a few basic or atomic elements in Python and each kind of element knows not only its value but also its type. It's very important to learn the difference between value and type. Python does not expect you to specify the type of a variable, but of course it runtime it has an exact type. As a programmer we must be aware of these types so that we don't try to divide two strings, for example.

Integers

These are just numbers that were used for counting; whole numbers like -9, 0, 3, 1023023823. What are the types/sizes?

```
>>> type(1000000000000000000)
<type 'int'>
>>> type(10000000000000000000)
<type 'long'>
```

How many bits to store?

```
>>> import math
>>> math.log(1000000000000000000, 2)
59.794705707972525
>>> math.log(10000000000000000000, 2)
63.116633802859894
```

You can make these things as big as you like:

```
>>> type(323241321234123412348123091324)
<type 'long'>
```

But if they are smaller they go into a different type. `type` is just another function call that asks the type of something.

```
>>> type(3232)
<type 'int'>
```

Real numbers

Real numbers, not whole numbers, are of finite precision but can hold some very large and very small numbers.

```
>>> 3.14159
3.14159
>>> 0.000000000001
```

```
1e-12
>>> 23e100
2.3e+101
```

The e stuff is the scientific notation and represents the exponent, not the mathematical constant *e*. We call these *floating-point numbers*.

The Python tutorial on floating-point numbers is something you should look at to learn more about floating-point numbers. The fact that they have finite precision, so-called "double precision," means you can get some odd results:

```
>>> 0.1 + 0.2
0.30000000000000004
```

This is because 0.1 is actually represented as 0.00011001100110011..., a repeating fraction. It has no nice representation in binary fractions, but numbers like 0.125 = 1/8 do: 0.001 in binary $= \frac{0}{2^1} + \frac{0}{2^2} + \frac{1}{2^3}$.

If you need floating-point numbers that trade precision for efficiency, use the decimal module:

```
>>> from decimal import Decimal
>>> Decimal(1)/Decimal(10) + Decimal(2)/Decimal(10)
Decimal('0.3')
```

One last thing on Floating-point numbers. *Be aware that subtraction can destroy precision*. It is considered an ill conditioned operation because subtracting two numbers that are almost equal can give you very imprecise answers.

Boolean values

A Boolean value is either true or false, but Python also allows a number of other things to represent true and false such as 1 can be true and 0 can be false.

```
>>> True
True
>>> False
False
>>> bool(1)
True
>>> bool(0)
False
>>> bool(36)
True
>>> bool("hi")
True
```

All of the comparison and logical operators yield Boolean types:

```
>>> 1 < 10
True
>>> 3.4 >= 10+0.2
False
```

```
>>> "cat" < "dog"
True
>>> 1 < 10 and "a"<"b"
True
>>> False or True
True
```

Strings

Single, double, or triple quotes. 0 or more characters in between delimiters. We call these literals or hard-coded values.

```
>>> 'a'  # a single character string is sometimes called a character
'a'
>>> 'hi'
'hi'
>>> "hi"
'hi'
>>> """hi"""
'hi'
```

When you need to actually include quotes of some kind in the string, then you surround it with different quotes like `"Bob's house"`.

Special characters

`\n` is the newline character, `\t` is the tab character

```
>>> print "Cars:\n\tBMW\n\tAudi"
Cars:
    BMW
    Audi
```

which is like doing:

```
>>> print "Cars:"
Cars:
>>> print "\tBMW"
    BMW
>>> print "\tAudi"
    Audi
```

or

```
>>> print "Cars:"
Cars:
>>> print "    BMW"
    BMW
>>> print "    Audi"
    Audi
```

Conversions

We can convert between numbers

```
>>> float(3)
3.0
>>> int(3.14159)
3
```

and numbers and characters

```
>>> chr(100)
'd'
>>> chr(105)
'i'
>>> ord('H')
72
>>> str(234)
'234'
```

What the types of operands.

```
>>> "hi" + 501
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: cannot concatenate 'str' and 'int' objects
>>> "hi" + str(501)   # convert number to string then add
'hi501'
>>> 2 / 3
0
>>> 2 / float(3)
0.6666666666666666
>>> 2 / 3.0
0.6666666666666666
>>> 10 == 10
True
>>> 10 == "10"
False
```