Understanding Data Part 1

Representing Data Digitally

Data

Data values can be stored in variables, lists of items, or standalone constants.

Computing devices represent data(text, numbers, audio, images) **digitally**, meaning that the lowest-level components of any value are **bits**.

Bit is shorthand for **binary digit** and is either 0 or a 1. This system of representing data is called **binary** (base 2) which uses only combinations of the digits zero and one.

```
name = "Mike Smith"        # represented internally as 0's and 1's
grades = [87, 91, 75]      # represented internally as 0's and 1's
```

Thus, **digital data** is simply some sequence of 0's and 1s representing some information. An image, audio recording or movie can be represented using just 0's and 1's!

Base 10 (Decimal numbers)

Computers store all information in the form of binary numbers. To understand binary numbers, let's first look at a more familiar system: decimal numbers.

Decimal numbers is base 10. It uses 10 digits {0,1,2,3...,9}. Why do we prefer base 10? There are many reasons but one simple reason is simply because we 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 16 (Hexadecimal) uses 16 digits: {0,1,2,3,4...,9,A,B,C,D,E,F}

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	

$3 \times 1000 + 2 \times 100 + 0 \times 10 + 7 \times 1$

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

$$\begin{array}{cccccc} 32 & 16 & 8 & 4 & 2 & 1 \quad \text{weight} \\ 1 & 0 & 1 & 1 & 0 & 1 \end{array}$$

$1 \times 32 + 0 \times 16 + 1 \times 8 + 1 \times 4 + 0 \times 2 + 1 \times 1$

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

Powers of 2

$2^0 = 1$
$2^1 = 2$
$2^2 = 4$
$2^3 = 8$
$2^4 = 16$
$2^5 = 32$

Subscripts Notation

Base 2

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

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

We'll use subscripts to denote the base of a number.

For example, 11_{10} is 11 in base 10. And 1011_2 is a number in base 2 that is equal to 11 in base 10. So,

$$1011_2 = 11_{10}$$

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

Here are all the 8-bit words:

$00000000_2 = 0_{10}$

$00000001_2 = 1_{10}$

$00000010_2 = 2_{10}$

…

$11111110_2 = 254_{10}$

$11111111_2 = 255_{10}$

Thus a byte is a string of length 8 whose value is either 0 or 1. A byte can be used to represent numbers in the range from 0 to 255.

For example, 10100110 is a single byte. We'll see later, for example, that a single byte can store a letter of the English alphabet.

Decimal to Binary(Method 1)

There are a couple of ways to convert from decimal to binary. This first method is a bit more intuitive.

Given a number n in decimal. We'll explain this process for n = 23. To convert to binary:

1) Make a table of powers of 2 from right to left starting at $2^0 = 1$. Choose the biggest power of two that will fit into the number you are converting. Your table will end with this number. For example, if n = 23, 16 is the greatest power of two that will fit into 23.

16	8	4	2	1

2) Write a 1 beneath the 16 in the table for the leftmost binary digit. Then, subtract 16 from your initial number. You now have 7.

Decimal to Binary(Method 1)

3) Using your new number (7), move down the chart marking how many times each power of 2 can fit into your dividend. 8 does not go into 7, so write a 0 beneath that box for the next binary digit to the right. Continue until you reach a number that can go into 7. Since 4 can fit into 7, mark it with a 1.

16	8	4	2	1
1	0	1		

4) Subtract 4 from 7 to get 3. Then repeat this for 3.

16	8	4	2	1
1	0	1	1	1

Thus $23_{10} = 10111_2$.

Decimal to Binary(Method 1)

Here's a visual example of how to convert 156 to binary.

Decimal to Binary(Method 2)

This method provides a nice algorithm for converting from decimal to binary. You can use this method, for example, to write a function in Python to do the conversion. On tests and worksheets, I recommend using Method 1.

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.

Answer: 10011101

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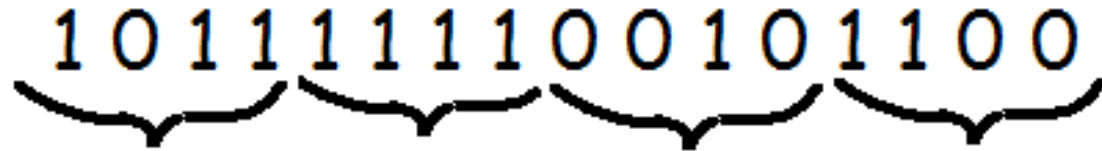
Hexadecimal digits: {0,1,2,3,4…,9,A,B,C,D,E,F}

Converting a binary number to a Hex number is relatively easy. Every 4 bits 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

1011 1111 0010 1100

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?

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

Overflow Error

In many programming languages, integers are represented by a fixed number of bits, which limit the range of integer values and mathematical operations on those values.

For example in Java, the range of the value of an integer is from −2,147,483,648 to +2,147,483,647. Trying to store a number bigger than the limits will result in an **overflow error**.

Some languages like Python, integers do not have limits on number size but, instead, expand to the limit of the available memory. Python floats are represented with 64 bits.

Overflow Error

The formula to calculate the largest number stored using n bits is $2^n - 1$.

Example 1:

With 4 bits, the largest integer that can be stored is $2^4 - 1 = 15$.

Example 2:

A 4-bit integer can any value in {0,1,2,...,15}. Thus, storing the value of 10 + 6 = 16 would cause an overflow error since the 16 = 10000 requires at least 5 bits.

Example 3:

If x is a 3-bit integer, then x = 111 + 111 will cause an overflow error since the sum is 1110 which requires at least 4 bits.

Round Off Errors

A fixed number of bits is used to store real numbers. Because of this limitation, round-off errors can occur.

Python computes 1/3 as 0.3333333333333333. This value is only an approximation of 1/3 which is an infinitely repeating decimal.

Round-off error occurs when decimals (real numbers) are rounded.

Abstraction

Abstraction is the process of reducing complexity by focusing on the main idea.

By hiding details irrelevant to the question at hand and bringing together related and useful details, abstraction reduces complexity and allows one to focus on the idea.

One type of abstraction we saw was **procedural abstraction**, which provides a name for a procedure(function) and allows it to be used only knowing what it does, not how it does it.

```python
import random
print(random.randrange(10)) # random number from 0 - 9
```

We don't need to know how the randrange() function is implemented to be able use it.

Abstraction

Bits are grouped to represent abstractions. These abstractions include, but are not limited to, numbers, characters, and color.

Sequences of 0's and 1's represent a string of characters, a numeric grade and a color in the examples below. The color example is done using Processing.

```
name = "Mike Smith"
grade = 87
c = color(255, 123, 110)
fill(c);
ellipse(250, 80, 100, 100)
```
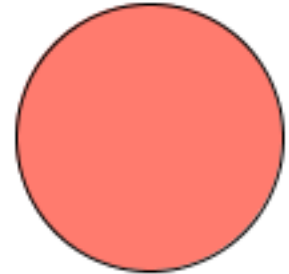
Encoding

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

To represent anything other than bits, we need 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. In ASCII, 65 represents A, 66 represents B and so on. The numbers are called code points. But ASCII only uses 8 bits. It does not have enough bits to encode other characters and languages(Chinese, French, Japanese).
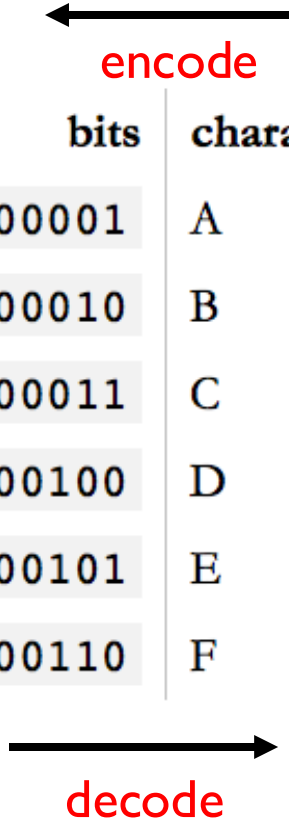
Similar to ASCII, Unicode provides a table of **code points** for characters: "*65 stands for A, 66 stands for B and 9,731 stands for* ❄*".* **UTF-32(Unicode Transformation Format), UTF-16 and UTF-8** are three encodings that use the Unicode table of code points.

Of the three, UTF-8 is by far the most widely used encoding. It is the standard encoding for all email and webpages(HTML5).

ASCII Sample

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

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* ☃".

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

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 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

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.

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.

Sequences of Bits

What does a list of numbers, an image and an audio file have in common?

They are all just lists of numbers!

The same sequence of bits may represent different types of data in different contexts.

```python
import numpy as np

import matplotlib.pyplot as plt

nums = np.loadtxt("numbers.txt")

print(nums)          # array([210 190 225 ..., ])

print(nums.size)          # 783126
```

These numbers can represent anything. For example, it could represent the number of characters in a list of 783,126 tweets that were tweeted over some one-minute interval.

Or this same list could represent something else entirely. See the next slide!

List of Numbers is An Image

The list of numbers in the previous example is now an image! We'll discuss how to write the code below in the next lecture.

```python
import numpy as np

import matplotlib.pyplot as plt

nums = np.loadtxt("numbers.txt", dtype="uint8")

print(nums) # array([210 190 225 ...,  ]), same list of numbers

nums = nums.reshape(417, 626, 3)

fig, ax = plt.subplots()

ax.imshow(nums)
```

The following output is produced by running the above code on the Jupyter Notebook

<matplotlib.image.AxesImage at 0x1045a8710>
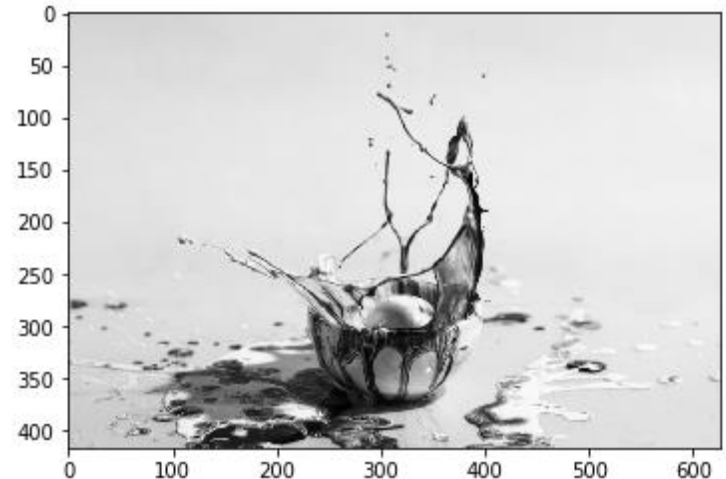
Digital vs. Analog

Computers can only understand **digital data**: discrete values like a list of integers, sequences of 0's and 1's.

Analog data have values that change smoothly, rather than in discrete intervals, over time. Some examples of analog data include temperatures over a period of time, pitch and volume of music, colors of a painting, or position of a sprinter during a race.

- analog signals are continuous and can take on an infinite possible values(the real numbers)
- digital signals are finite. For example, 8-bit colors can take on one of 256 discrete, finite possibilities. But actual colors can take on any of an infinite possible values or shades.

Digital vs. Analog

Analog data can be closely approximated digitally using a **sampling technique**, which means measuring values of the analog signal at regular intervals called **samples**.

The samples are measured to figure out the exact bits required to store each sample. The number of samples measured per second is the **sampling rate**, the higher the rate the better the quality.

CD-quality has a rate of 44,100 samples per second(44,100 Hz or 44.1 kHz).
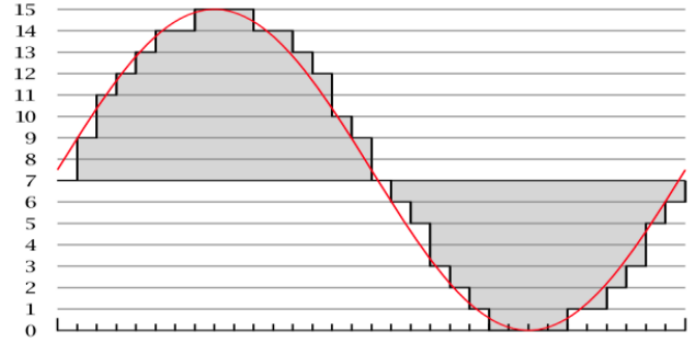
Analog   Low sampling rate   High sampling rate