Data Representation

Instructor Guide

Overview

The lab will explore data representation, or how standard pieces of data such as numbers, images, and strings are stored on a computer. It will cover binary, bitwise operators, masks, ASCII, and Morse Code.

Learning Goals

- Understanding binary, why it is useful, binary patterns, and how numbers are represented on a computer
- Understanding bitwise operations, how images are represented on a computer, and image masks
- Understanding ways letters are represented on a computer and in binary (ASCII, Morse Code, and Braille).
- Learning how to record audio on a computer

Personal Growth Goals

- Teamwork: They need to divide up the work of recording audio in Activity 3 across the class.

- Empowerment: Students will realize that they have the skills necessary to help underprivileged populations (through the Braille Reader in Activity 3)

Skills Required

- A brief exposure to 2D lists (for Activity 2)
- A brief exposure to dictionaries (for Activity 3)

Resources Required

- One computer per student with Python 3 installed, programming files loaded
- An audio-recording device (can be the computer, or a smartphone)
- One or more instructors to lead the class, and one mentor per two students to work together in small groups on the activities

Instructor Preparation

- Determine a prize for the winner(s) of the scavenger hunt
- Print and cut out these binary index cards, one set per student

In Depth Description of Lab Activities

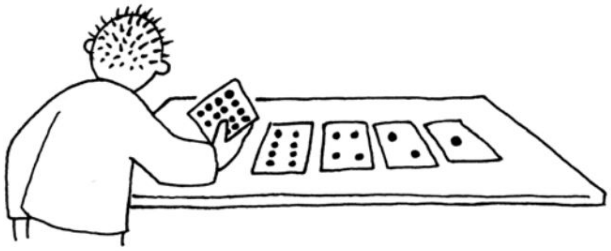
Part 1: Binary

Begin the class by leading a discussion about why we count things in multiples of 10. A decade is 10 years. A dollar is 10 dimes, and 100 cents. A meter is 100 centimeters. But why 10? After discussing this, ask the students to think about a hypothetical creature with only one finger on each hand. How would that creature count? They can clearly count up till two, but then they have to remember the 2 and start over. They can then count up till 4, but then they have to remember they have two 2s and start over. And so on. But as you get to larger numbers, wouldn't it be useful to write down how many of each number you already have? This is where binary comes in.

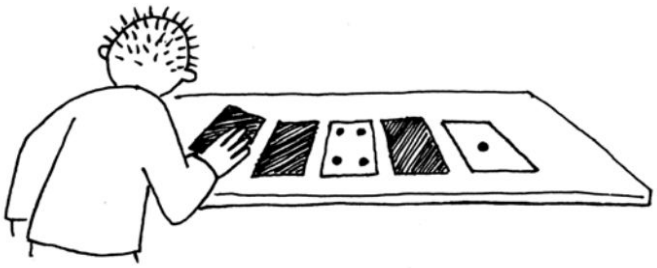
Briefly go over binary with the students. Then have them do the following activity:

Activity 1: Scavenger Hunt

This activity will be done individually by each student, with one mentor working with an individual student or small group of students to help them through the activity.



Lay the 5 binary index cards out, one set in front of each student. Make sure the cards are placed in exactly the same order. Now have the students flip the cards so exactly 5 dots show—keep them cards in the same order!



Find out how to get 3, 12, 19. Is there more than one way to get any number?

Now try to represent the numbers 1 through 8 in order. Do you notice a pattern?

Now have the students each pick a number and represent it using the cards (don't tell them what we are doing next). Once they have all represented a number, it's time for a scavenger hunt! Each student will be given a paper, with criterion. For each criterion, then have to find at least one number that satisfies it, and write the name of the student that wrote that number next to the criterion. Students that get the most criterion will get a reward (TBD)! Here are sample criterion:

After the scavenger hunt, lead some discussion on patterns the students noticed. For example, what are easy ways to find even/odd numbers? Why is it that in decimal (base 10) it is so easy to find multiples of 5, but not in binary (base 2)?

Optional Activity 1: Finger Counting

Some people may think why count using binary. Can't you count less numbers using a base-2 system than a base-10 system? Well, not exactly. Let's take your fingers for example. In the base-10 system, we can only count to 10, right? With base-2, you can use your fingers to count above 1,000! Here's how:

1. Make your right-most finger the ones place, and then increase the places by a factor of 2 moving left. For example, if my palms are facing up, my right thumb would represent the 1s digit, my right index finger would represent the 2s digit, my right middle finger will represent the 4s digit, etc.

2. When a finger is up, that means you have a 1 in that digit place. When a finger is down, you have a 0 in that place. When a finger is up, you have a 1 in that place.
3. Have the students make some specific numbers. Ask them what the largest number they can make is. Ask them what number popular hand signs (such as the V with the index and middle fingers up, and the sign (is it called fox ears?) with the thumb, index finger, and pinky up, represent.)

Part 2: Bitwise Operators & Images

Now move into bitwise operators. Start by explaining to students that just like you can add with base-10 numbers, you can also add with base two numbers. Show a couple of of examples, and have the students do the conversion from base-10 to binary to verify that addition is the same.

After teaching addition, teach OR & AND. Motivate this by talking about logical statements in grammar. “Barack Obama is a president AND pigs fly.” “Carnegie Mellon is an elementary school OR Dave and Andy’s serves ice cream.” Are those statements true or false? Why? What is the relationship between the sub statements and the whole statement?

Then talk about the view of binary where 1 is true and 0 is false. When interpreted that way, OR & AND make a lot of sense. Talk about how you OR/AND together larger numbers by OR/AND together the individual digits (be sure to mention the leading 0s). Discuss patterns in OR and AND-ing. What happens when you AND a number with 1? What if you OR/AND with the same number twice? What if you AND with a 0?

As students might notice, when you AND a digit with a 0 it becomes 0, and when you OR a digit with 1 it becomes 1. This is useful in “nullify”-ing pieces of data. For example, if you want to remove the bottom 4 digits of a number (which will round down to the nearest multiple of 16) you would AND the number with 1...10000.

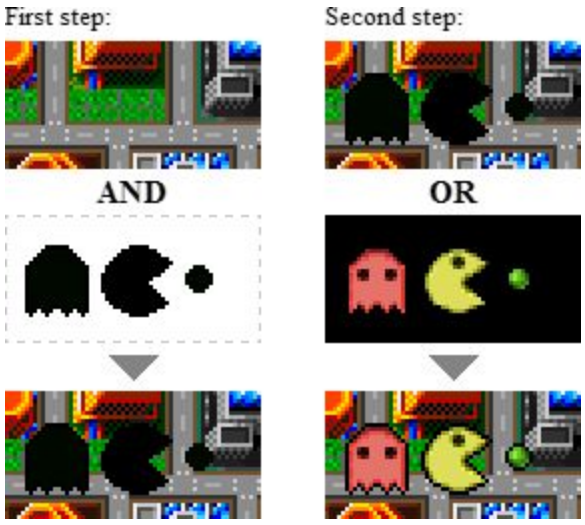
Activity 2: Image Masks

How many students have played or heard of Pokemon Go? In it, you can catch Pokemon in AR, or augmented reality mode, where your Pokemon is projected on a picture of your real surroundings. How does the game put pokemon there?

In addition to nullify-ing parts of a number, you can also nullify parts of an image. Lead a class discussion about what an image is -- what characterizes it and what are its unique properties. Then talk about how an image is represented inside a computer: a 2D list of (r,g,b) values, where each indicates the amount of that color in the pixel (each number is between 0 and 255). Therefore, (0,0,0) is white, and (255,255,255) is black. Hence, if we OR a pixel with white, it stays the same. And if we AND it with black, it stays the same.

Using this concept, can we superimpose two images? Show students two images (such as the ones below) and ask how ***they*** might put Pikachu on top of the city (possible answers might be cutting it out and pasting it on top, using an image-editing software, etc.)

Then talk about how a computer effectively cuts certain parts of an image out, except the way it does that is by using masks. It makes another image that is as large as the current one, with only white and black. White in the places you want to keep, and black in the places you don't. It then and's them with each image. You now have two images with black in exactly opposite areas. You then take those two images, and or them together. For example, the steps below.



Working with mentors, students will code up a way to image mask. They will use helpers in imageMaskHelpers.py, which allow them to open an image as a 2D list of pixels, and write a 2d list of pixels as an image. Mentors should go over questions like the following with them:

1. What are ways to represent a mask?
2. What information will the functions need as input?
3. What are sample pictures you might want to mask?
4. What if, instead of masking, we wanted to take certain colors out of a picture? (For example, making it a red-scale picture). How would we do that?

Part 3: Representing Letters

Now move into how how letters are represented by pointing out that most students in the class probably don't use their computers primarily for image editing or calculations, but rather to write essays, emails, Facebook Messages, etc. So how does a computer represent the 26 letters of the English language, both capital and lowercase, and the symbols (commas, quotation marks, etc.) that are commonly used?

Well what if we assign one letter to each number? For example, we can assign capital A to 0, capital B to 1, etc. And then once we are done with the capital letters, we can assign lowercase a to 26, lowercase b to 27, etc. all the way up to 55. But what about things like spaces, enters, or tabs? And how do we tell a computer when a file is done? In order to account for this, members of the American Standard's Association developed ASCII in the 1960s, which is exactly what we discussed: a way of pairing up letters and other characters used in text to numbers in order to represent them on a computer. In ASCII, the first 32 characters are reserved for "control characters", or characters that tell a computer how to read the text. For example, start of file, end of file, enters, tabs, etc. Then we move into symbols like exclamation mark, question mark, etc. and then eventually into uppercase and lowercase letters.

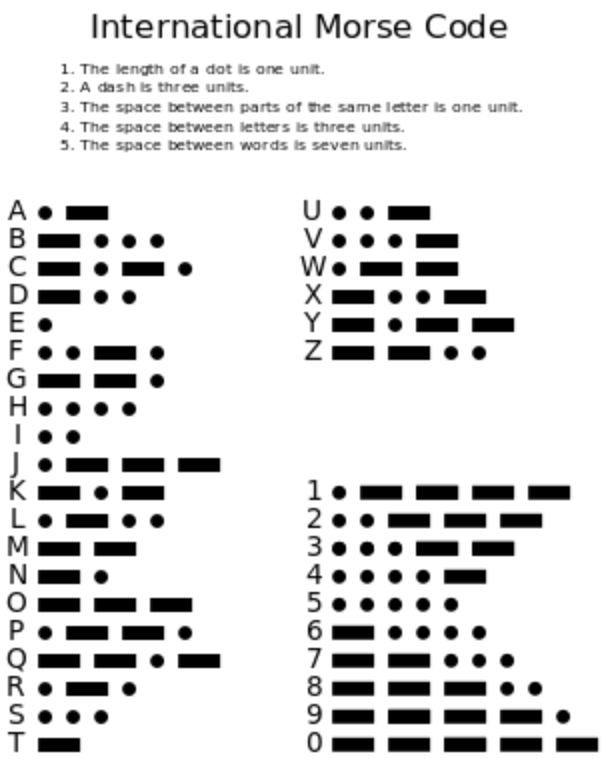
0	<NUL>	32	<SPC>	64	@	96	`	128	Ä	160	†	192	¿	224	‡
1	<SOH>	33	!	65	A	97	a	129	Å	161	°	193	¡	225	·
2	<STX>	34	"	66	B	98	b	130	Ç	162	¢	194	¬	226	‚
3	<ETX>	35	#	67	C	99	c	131	É	163	£	195	√	227	„
4	<EOT>	36	$	68	D	100	d	132	Ñ	164	§	196	ƒ	228	‰
5	<ENQ>	37	%	69	E	101	e	133	Ö	165	•	197	≈	229	Â
6	<ACK>	38	&	70	F	102	f	134	Ü	166	¶	198	Δ	230	Ê
7	<BEL>	39	'	71	G	103	g	135	á	167	ß	199	«	231	Á
8	<BS>	40	(	72	H	104	h	136	à	168	®	200	»	232	Ë
9	<TAB>	41	)	73	I	105	i	137	â	169	©	201	…	233	È
10	<LF>	42	*	74	J	106	j	138	ä	170	™	202		234	Í
11	<VT>	43	+	75	K	107	k	139	ã	171	´	203	À	235	Î
12	<FF>	44	,	76	L	108	l	140	å	172	¨	204	Ã	236	Ï
13	<CR>	45	-	77	M	109	m	141	ç	173	≠	205	Õ	237	Ì
14	<SO>	46	.	78	N	110	n	142	é	174	Æ	206	Œ	238	Ó
15	<SI>	47	/	79	O	111	o	143	è	175	Ø	207	œ	239	Ô
16	<DLE>	48	0	80	P	112	p	144	ê	176	∞	208	–	240	
17	<DC1>	49	1	81	Q	113	q	145	ë	177	±	209	—	241	Ò
18	<DC2>	50	2	82	R	114	r	146	í	178	≤	210	"	242	Ú
19	<DC3>	51	3	83	S	115	s	147	ì	179	≥	211	"	243	Û
20	<DC4>	52	4	84	T	116	t	148	î	180	¥	212	'	244	Ù
21	<NAK>	53	5	85	U	117	u	149	ï	181	µ	213	'	245	ı
22	<SYN	54	6	86	V	118	v	150	ñ	182	∂	214	÷	246	ˆ
23	<ETB>	55	7	87	W	119	w	151	ó	183	∑	215	◊	247	˜
24	<CAN>	56	8	88	X	120	x	152	ò	184	∏	216	ÿ	248	¯
25	<EM>	57	9	89	Y	121	y	153	ô	185	π	217	Ÿ	249	˘
26	<SUB>	58	:	90	Z	122	z	154	ö	186	∫	218	/	250	˙
27	<ESC>	59	;	91	[	123	{	155	õ	187	ª	219	€	251	°
28	<FS>	60	<	92	\	124	|	156	ú	188	º	220	‹	252	¸
29	<GS>	61	=	93	]	125	}	157	ù	189	Ω	221	›	253	˝
30	<RS>	62	>	94	^	126	~	158	û	190	æ	222	fi	254	˛
31	<US>	63	?	95	_	127	<DEL>	159	ü	191	ø	223	fl	255	ˇ

But how do we tell a computer when one letter ends and the other starts? For example, in ASCII H is 72, i is 105, and ! is 33. In other words, Hi! would be **1001000**1101001**100001**. But how does the computer know when the H ends, the i starts, and the exclamation mark starts? For example, it could read the above as **10010**01101**10011**00001, which is <DC2><ACK><DC3><NUL>? It sounds ridiculous, but remember, a computer doesn't know English, so it can't check what combination of characters makes sense or not. Plus, sometimes people want to type gibberish! So how can a computer know how to read this long chain of 0s and 1s?

The way ASCII solves this problem is by having each number exactly 8 bits (a bit is a digit of binary, either 0 or 1) long. That way, the computer can read 8 bits at a time, and convert each one to a letter, and there is no confusion of where to start or end reading a letter. Although it seems like too little, note that each digit in binary represents a new power of two. So with 8 bits, we can actually write numbers all the way up to 2^8-1, or 255. That is more than enough for English!

However, the downside of ASCII is that every number takes up exactly the same amount of space (8 bits). That is not particularly easy for communication! Imagine trying to communicate with your friend who lives across the street by shining lights through a window. You could say that when the light is on, that represents a 1, and when it is off that represents a 0. But then for every letter, you would have to turn it on and off 8 times. Is there any way to make it easier?

Well, what if we rearrange the letters. Why should A be 65, when we use it a lot more than some of the control characters? If we order letters by the most frequently used letters in English, we get a very different, and much more compact system. E would be 0. T would be 1. And so on.

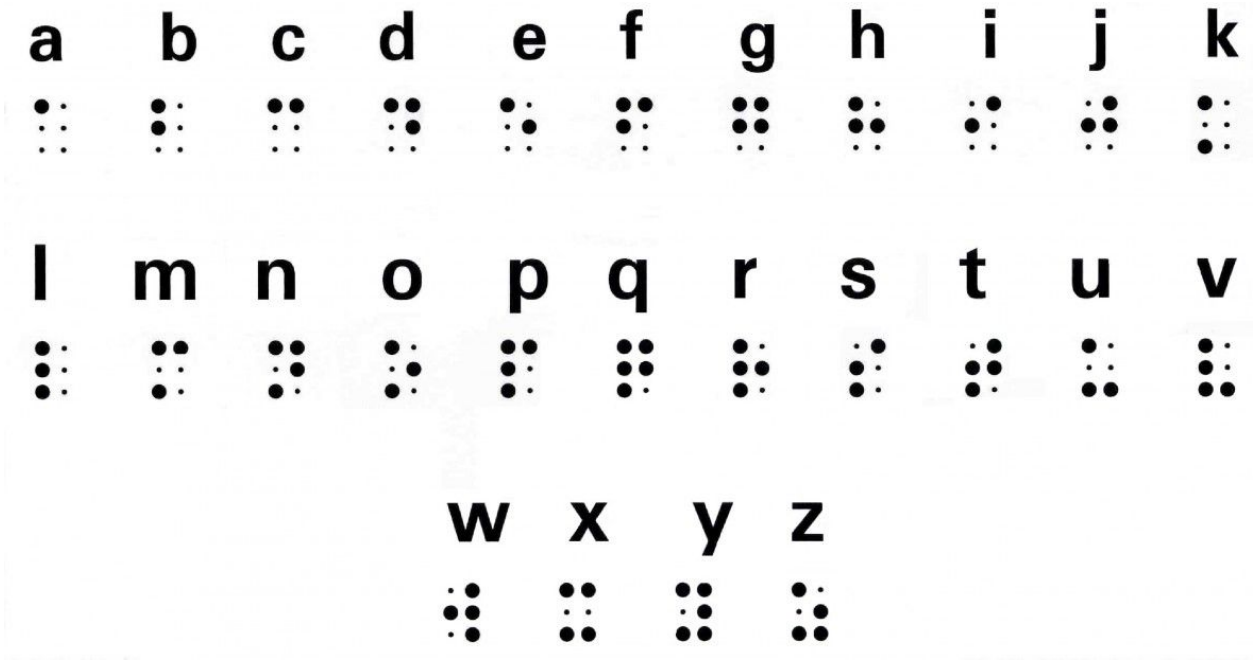


In fact, this is exactly what Morse Code does! Morse code is not used by computers, but it is a way to represent letters with a binary system of “dots” and “dashes”. It was used starting in the 1800s for a variety of tasks: sending messages long-distance (telegraph), communicating between boats and/or planes (through radio or light), and making communication more accessible to people with severe motor disabilities. In morse code, dots represent a short burst of something, and dashes represent a longer burst. Between letters, you leave a longer break to indicate that you are moving on to the next letter, and between words you take an even longer pause (like a space.) But the best part of Morse Code is you can send it in any way and in many different places, even by tapping your pencil on your desk! However, it is important to note that Morse Code is only one crack at trying to spend less space, or number of bits, representing numbers in binary. Many others exist, such as Huffman Encoding, which is similar to the method used to create compressed .zip or torrent files on your computer.

Activity 3: Braille Reader

ASCII and Morse Code are just two examples of ways to represent letters using binary. Another is Braille, a writing system for blind and visually impaired people. People who are blind can’t read or write letters, so instead they rely on another written alphabet made of bumps. Braille, which was invented by Louis Braille in the 1800s, is a system of writing where each letter is

represented by bumps in a sheet of paper. A letter consists of a 2 by 3 grid of dots, where some are indented and some are not. This is basically a system of 6 0s and 1s, where a 1 indicated a bump.



However, Braille was a bit harder to design than ASCII, because in addition to finding a unique number per letter, they had to make sure it was unique both when read rightside-up and upside-down. Because if a letter meant one thing when read right-side up and another thing when read upside down, how would a blind individual be able to tell which side of the paper is up? Hence, even if you rotate all the Braille letters 180 degrees, there is no overlap amongst them (except for x, which is pretty uncommon).

However, for blind individuals that grow up writing in Braille, writing on a QWERTY keyboard must be pretty confusing at first. How about we write a program that allows them to type Braille on a computer, and then plays them back the letter they typed? Use the helper functions in brailleReaderHelpers.py to write a program that allows users to use the standard keyboard to write in Braille. The package has helper functions that allow the students to read key-presses, and play sounds. The students will have to work with their mentors and classmates to determine the following:

1) What keys do they want to map to what dots in a Braille cell?
2) How often do they want to play back audio, and what audio will they play back? (We will have a shared Dropbox folder that students can upload their audio to, so they can divide up the recording work).
3) How will they represent binary letters on the computer?

Optional Activity 3: Card Trick

Binary is not only useful for representing numbers and letters, but can also be used for card tricks! Take a standard deck of 52 cards, and pick a student to lay out 25 of them in a 5x5 grid (25 cards). They can randomly decide which cards to place face-up and which to place face down. Don't look while they are placing cards. Then, tell them: "In a second, I will ask you to flip a card over. But before that, let me add an extra row and column in order to make it harder for myself." Then, go ahead and add an extra 11 cards (to make it a 6x6 grid). However, in order to determine whether to lay a card face-up or face-down, count how many cards are face-up and face-down in that row/column already. If there are an odd number of face-up cards, make your card face-up. If there are an odd number of face-down, make it face-down. Basically, after you add your cards, you want both the number of face-up and face-down cards in every row/column to be even. Note that this will be possible for every card you place except for possibly the last one (since it will complete both a row and a column). If after placing the last one, one row or column has an odd number of face-down or face-up cards, remember just that row or column.

Now, turn your back and let the student turn exactly one card over. Then, turn back around and try to find their card. Since before the student flipped the card, every row and column (except possibly one) had an even number of face-up/face-down cards, the card that was flipped will be in the row and column that has an odd number of face-down/face-up cards! (If the card was in the one row/column with an odd number, then by flipping it the row/column would have gone from an odd number to an even number).

Show this card trick to students and let them try to figure it out for a bit. Eventually, share how to do it with them, and let them practice on their friends if they so desire.

NOTE: this card trick can be done with any size of grid. For example, if you have two decks (104 cards) you can start with a 9x9 grid, and add a row/column to make it 10x10. The only catch is the original grid needs to have an odd number of rows and columns.

Happy tricking!

Lesson Plan

(:10) means that this part should be done by the tenth minute of the lesson

1. Setup (:5)
2. Part 1 (:15)
3. Activity 1 (:30)
4. Part 2 (:45)
5. Activity 2 (:75)

6. Part 3 (:90)
7. Activity 3 (:120)