Bonus Slide

CSD: Computer Science Department

CMU's CSD was created in 1965, originally in MCS (but now in SCS). Computer science came from math, and much work in this department involves using math to prove computational properties.

People in CSD study the theory of algorithms, the design of programming languages, how hardware and software work together at scale, AI, security, graphics, and more.

Teaching AI-Generated Scenes To Obey Physics

Using Computer Science To Save the Bees

Unit 4: Demystifying CS

Demystifying Computers:

How They Communicate and Represent Data

15-110 – Friday 08/28

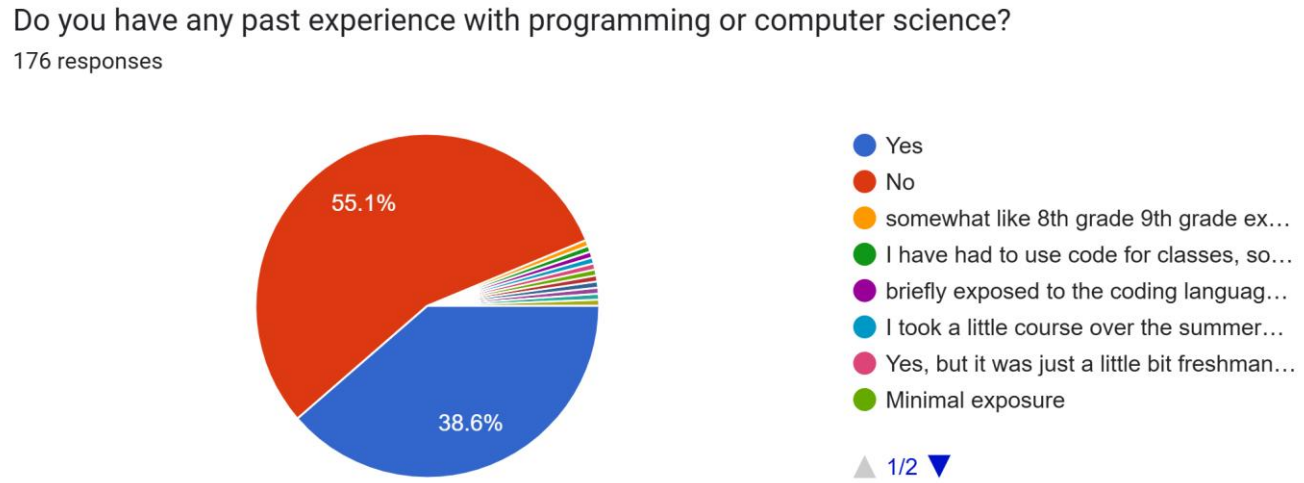
Announcements

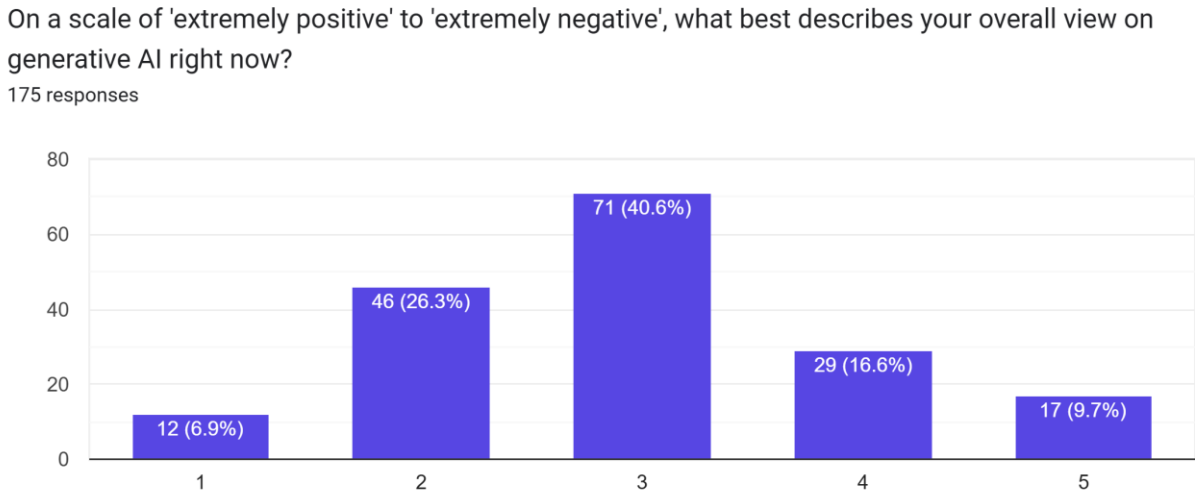
- Hw1 is due **Monday at noon**

- 62% of students have submitted the Practice Set – Week 1 (P1).
  - If you haven't, make sure to submit your work on Gradescope by the Monday 2pm deadline

Announcements – Pre-Semester Survey

- 176 responses so far (76%)
- 55% report no prior knowledge
- Mix of views about generative AI





Learning Objectives

- Use **abstraction** to communicate about a system with an appropriate level of detail.

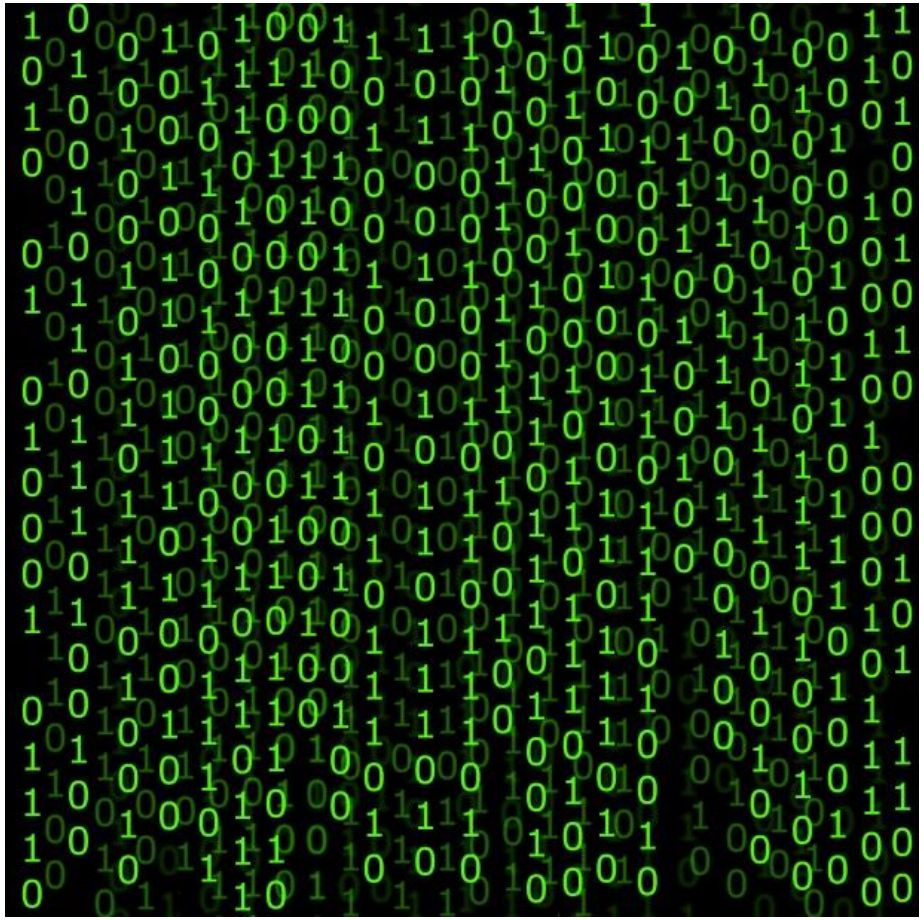
- Use **binary numbers** to represent varying types of data stored on a computer.

Computers Run on 0s and 1s

Computers rely on just two symbols, 0s and 1s, both to **represent** information and to **communicate** it.

You've likely seen references to this before in media.

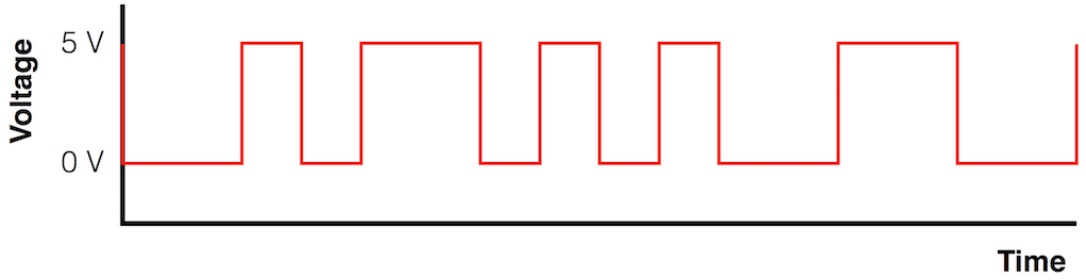
In computing, we call 0s and 1s **bits**.



Why 0s and 1s?

In hardware, bits are represented as **electrical voltage**. A high level of voltage is considered a 1; a low level of voltage is considered a 0.

Changing the voltage at different points in the system is how computers store and manipulate data.



Abstraction is About Representation

How do computers speak in 0s and 1s? How do they understand what we tell them, and how can text, images, or sound all be represented with just 0s and 1s?

This brings us back to **abstraction**. Recall how we discussed abstraction in the first lecture:

> *In computing, we use abstraction to make complex procedures and systems manageable by adapting the level of detail provided.*

We'll use abstraction to facilitate communication with computers and interpret 0s and 1s in a wide variety of ways.

Part 1:
How Computers Communicate

Ways of Communicating

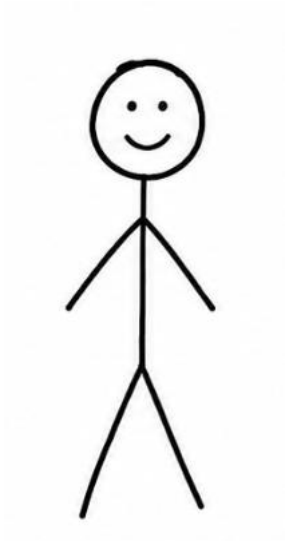
Go to the Python shell in Thonny, type `3 + 5` and press Enter.
What do you notice happens?

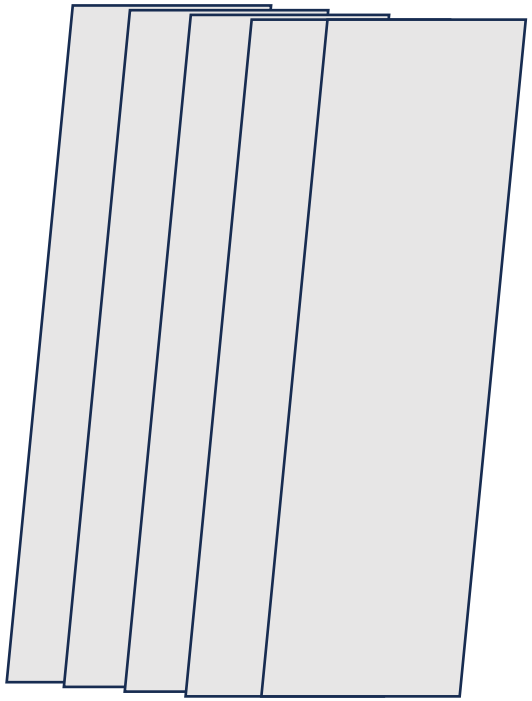
Now click the X button on the Thonny window to close the application.
What happened?

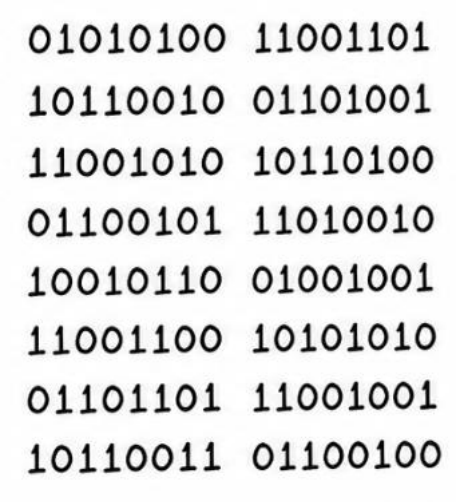
Both are ways of communicating with the computer, one through a program and the other through interaction (a click). Yet both get translated into 0s and 1s underneath!

How does that translation actually happen?

Layers of Abstraction







Human Intent:
Add 3 and 5

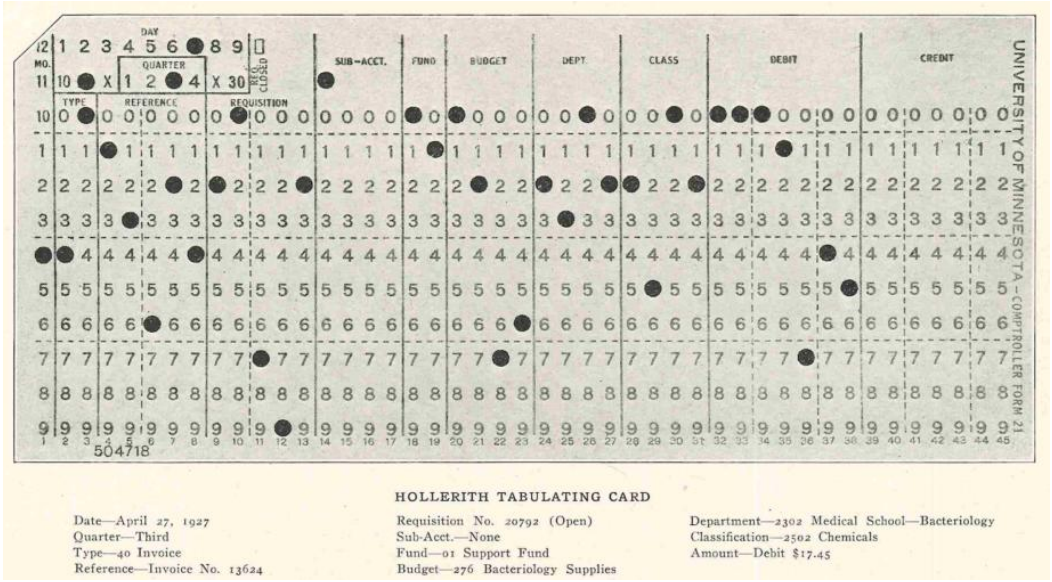
Multiple layers of abstraction
remove us from 0s and 1s

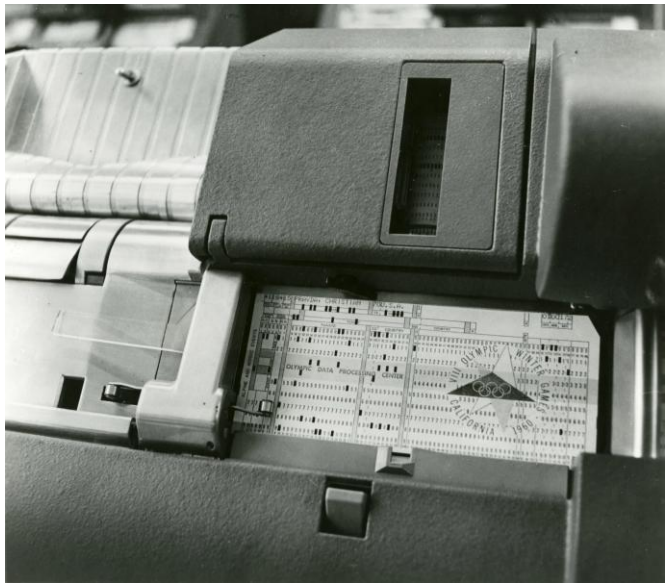
Punch Cards

Originally instructions were physically encoded as holes in stock; a hole completed an electrical circuit (1), no hole meant no electrical circuit (0).

Little abstraction but no convenience; one wrong hole, or one dropped box of cards, could destroy the program.

We weren't making computers talk like us; we were talking like computers, directly in 0s and 1s.





Assembly

Assembly is a low-level language with a near 1:1 correspondence between its text statements and the 0s and 1s in the hardware operations.

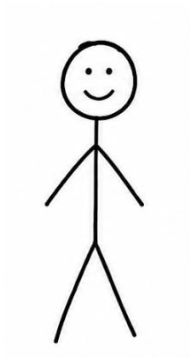
Instead of writing raw 1s and 0s, programmers use human-readable abbreviations called **mnemonics** (like MOV, ADD, or JMP) to control hardware.

This is our first real layer of abstraction! Still far from "talking like us", but a mid-level of abstraction above punch cards.

```
01 .MODEL SMALL
02 .STACK 100H
03 .CODE
04
05 MOV AX, 0x3C
06 MOV BX, 0000000000001010B
07 ADD AX, BX
08 MOV BX, 14
09 SUB AX, BX
10
11 MOV AH, 04CH
12 INT 21H
```

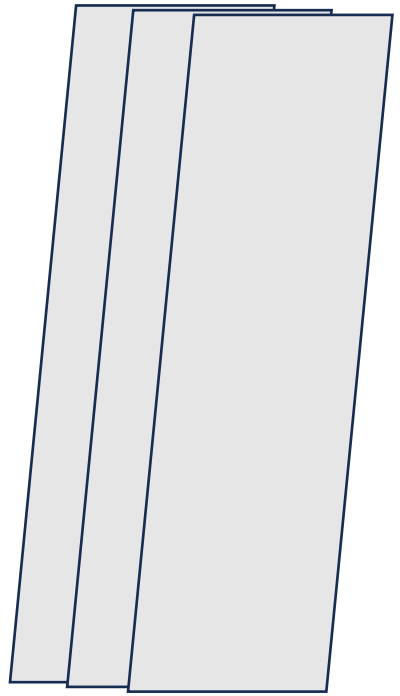
100% deterministic
Same code, same input == same output every time

Layers of Abstraction



Writes assembly

```
push    rbp
mov     rbp, rsp
sub     rsp, 16
mov     eax, 3
add     eax, 5
mov     DWORD PTR [rbp-4], eax
leave
ret
```



```
01010100 11001101
10110010 01101001
11001010 10110100
01100101 11010010
10010110 01001001
11001100 10101010
01101101 11001001
10110011 01100100
```

Human Intent:
Add 3 and 5

More layers of abstraction remove us from 0s and 1s

High-level programming languages

In our quest to make computers speak more like us, we built another layer of abstraction: high-level languages (like Python, Java or C++).

High-level programming languages are designed with strong abstraction to be easily read, written, and maintained by humans. They use English-like phrases and mathematical symbols, hiding complex hardware operations.

A **compiler** or **interpreter** translates high-level programming code down to assembly.

100% deterministic
Same code, same input == same output every time

Layers of Abstraction

Python Code:

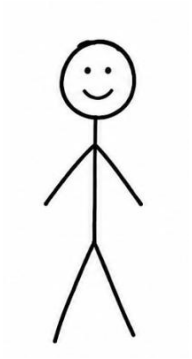
```
1  3 + 5
2
```

Compiler/
Interpreter

```
push    rbp
mov     rbp, rsp
sub     rsp, 16
mov     eax, 3
add     eax, 5
mov     DWORD PTR [rbp-4], eax
leave
ret
```

Assembly

```
1011100000000011
0000000000000000
0000000000000101
0000010100000000
0000000000000000
```



You write
the code

Human Intent:
Add 3 and 5

**Note, there are other layers of abstraction in between!!

The Computer Does What You Say, Not What You Intend To Do!

Python Code:

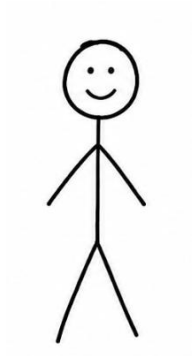
```
1 3 + 5
2
```

Compiler/ Interpreter

```
push    rbp
mov     rbp, rsp
sub     rsp, 16
mov     eax, 3
add     eax, 5
mov     DWORD PTR [rbp-4], eax
leave
ret
```

Assembly

```
1011100000000011
0000000000000000
0000000000000101
0000010100000000
0000000000000000
```



You write the code

Human Intent:
Add 3 and 5

- Computers execute your instructions exactly as written; they have no sense of what you *intended*, only what you *typed*
- This is why we need to verify the code we write by asking "*what did I actually tell it to do?*" rather than "*why isn't it doing what I meant?*"

GenAI: A shift from deterministic to probabilistic computing

What is GenAI?

GenAI stands for Generative Artificial Intelligence, a type of system that can create text, images, audio and even code, by learning patterns from huge amounts of existing data.

GenAI is Non-deterministic

Python Code:

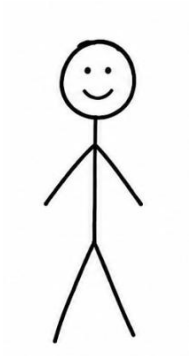
```
1  3 + 5
2
```

Compiler/ Interpreter

```
push    rbp
mov     rbp, rsp
sub     rsp, 16
mov     eax, 3
add     eax, 5
mov     DWORD PTR [rbp-4], eax
leave
ret
```

```
1011100000000011
0000000000000000
0000000000000101
0000010100000000
0000000000000000
```

GenAI writes code



You write the prompt

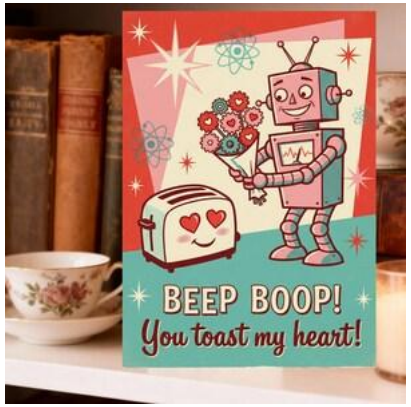
"I want to add 3 and 5 in Python. How do I do that?"

Human Intent:
Add 3 and 5

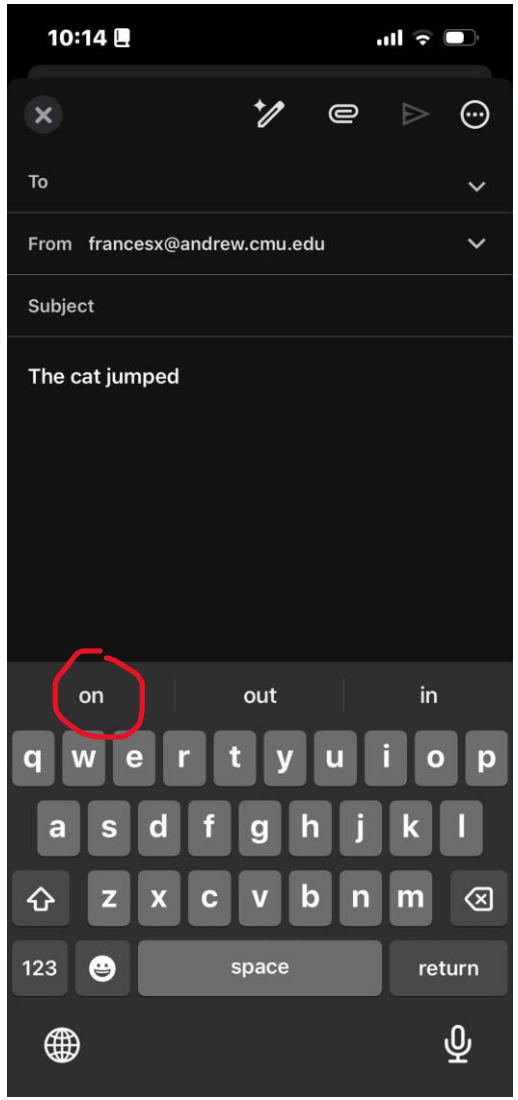
- GenAI is non-deterministic! The same prompt, run multiple times, even on the same model, can produce different results.
- The code GenAI writes can vary based on your prompt, and it may even be **wrong!**

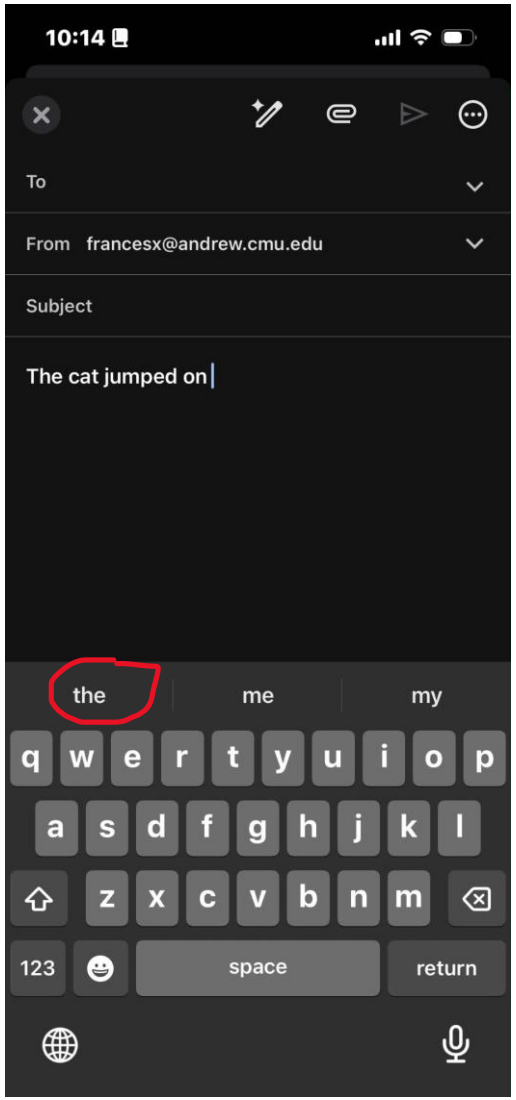
Poem example:

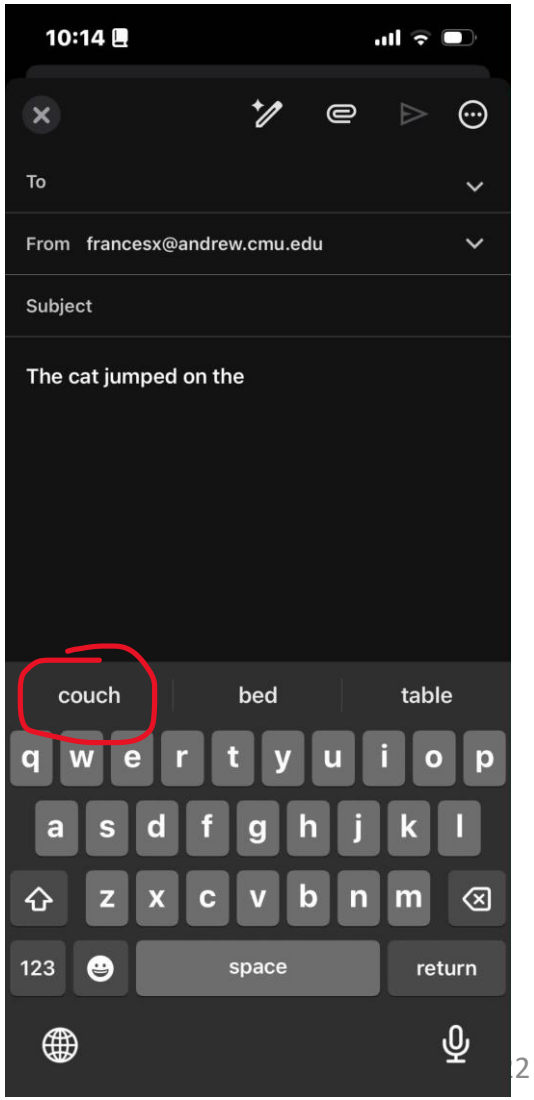
Prompt: *Write a quick four-line poem about a data-tracking robot that accidentally falls in love with a toaster. Do not use the word 'beep'.*



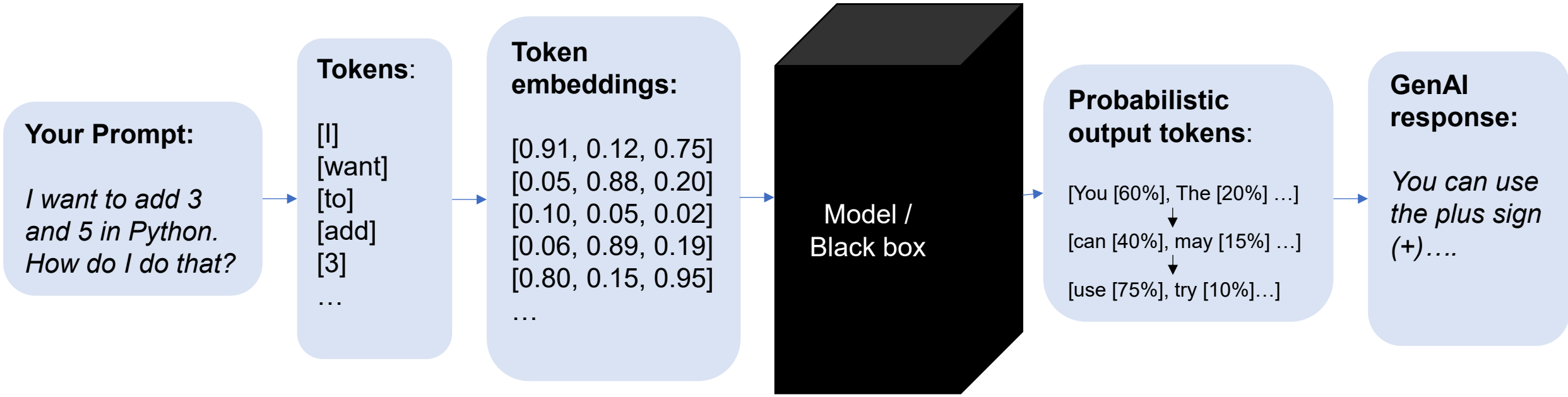
Analogy: Phone Autocomplete







Why are GenAIs Non-deterministic? How do they work?



- **Prompt**: The input you give a GenAI system, a question, instruction, or description of what you want.
- **Token:** A small chunk of text, often a word, used by GenAI to understand your prompt.
- **Token embeddings:** Words turned into numbers to capture their meaning.
- A **model:** the underlying system that a GenAI uses to generate responses; trained on massive amounts of data to recognize patterns and predict the most probable next token.

Summary:

100% deterministic
Same code, same input == same output every time

Python Code:

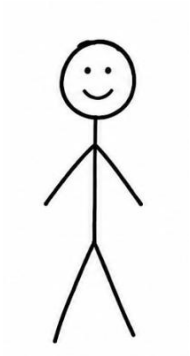
```
1  3 + 5
2
```

Compiler/
Interpreter

```
push    rbp
mov     rbp, rsp
sub     rsp, 16
mov     eax, 3
add     eax, 5
mov     DWORD PTR [rbp-4], eax
leave
ret
```

```
1011100000000011
0000000000000000
0000000000000101
0000010100000000
0000000000000000
```

GenAI
writes
code



You write
the prompt

"I want to add 3 and 5 in Python. How do I do that?"

Non-deterministic!
The same prompt, run multiple times, even on the same model, can produce different results.

Human Intent:
Add 3 and 5

Part 2:
How Computers Represent Data

Computers Run on 0s and 1s

How do computers create a 1:1 correspondence between assembly and 0s and 1s?

How can they translate *everything* to 0s and 1s?

Let's return to abstraction. If we can identify what is **most important** about a type of information, we can **map it** to 0s and 1s to create a representation.

Example: Assembly

In assembly, the most important part is the core action that needs to be taken. Each human-readable instruction is mapped to a unique set of 0s and 1s.

Add into a register	00000000
Move data to new register	10001000
Compare two values	00111000
Jump to a new instruction if a condition is met	01110000

https://en.wikipedia.org/wiki/List_of_x86_instructions

Binary Numbers Use 0s and 1s

How can we identify all possible combinations of 0s and 1s that can be used as codes?

These can be organized using a **number system**, like the decimal numbers we use everyday. Decimal numbers use the digits 0-9. Here we'll introduce **binary numbers** which only use the digits 0-1.

Number Systems - Decimal

In a decimal number, moving from the right, the first digit is the number of 1s, the second is 10s, the third is 100s, etc. Each digit represents a **power of 10**.

For example, 1980 in decimal is 1 * 1000 + 9 * 100 + 8 * 10 + 0 * 1.

10^3	10^2	10^1	10^0
1000	**100**	**10**	**1**
1	9	8	0

Number Systems – Binary

For a binary number, count the number of 1s, 2s, 4s, 8s, etc. Why these numbers? They're **powers of 2**. This is a number in **base 2,** which only needs the digits 0 and 1.

For example, 1101 in binary is 1 * 8 + 1 * 4 + 0 * 2 + 1 * 1 = 13 in decimal.

2^3 **8**	2^2 **4**	2^1 **2**	2^0 **1**
1	1	0	1

Counting with Binary

When we count in decimal, we add new digits from **right to left:**

0, 1, 2, 3, 4, 5, 6, 7, 8, 9…

10, **1**1, **1**2, **1**3, **1**4, **1**5, **1**6, **1**7, **1**8, **1**9…

20, **2**1, **2**2, **2**3, **2**4, **2**5, **2**6, **2**7, **2**8, **2**9…

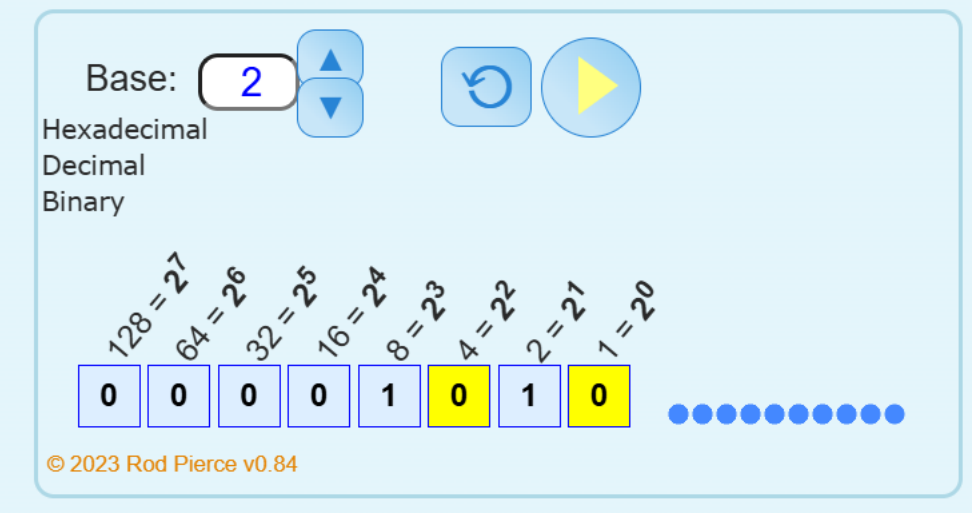
When counting in binary, do the same thing, but **only using 0 and 1**.

0, 1…

10, **1**1…

100, **1**01,

1**1**0, 1**1**1…



https://www.mathsisfun.com/binary-number-system.html

Big Idea: Numbers are Numbers

Importantly, the quantities represented by binary numbers are the **same** as the quantities represented by decimal numbers.

That means every binary number can be translated to a decimal number, and vice versa. It's just another way of counting.

10010 -> 16 + 2 = 18

35 = 32 + 2 + 1 -> 100011

You can convert numbers using the calculator here: https://www.calculator.net/binary-calculator.html

Bits and Bytes

When working with binary and computers, we often refer to a specific number of binary values used together to represent some piece of information.

A single binary value is called a **bit**.

A set of 8 bits is called a **byte**.

We commonly use some number of **bytes** to represent data values.

Abstracted Types

Binary and Abstraction

Now that we understand how binary numbers work, we can represent **everything** computers store using binary.

We just need to use **abstraction** to interpret a set of bits or bytes in a particular way.

Let's consider dates, images, and text.

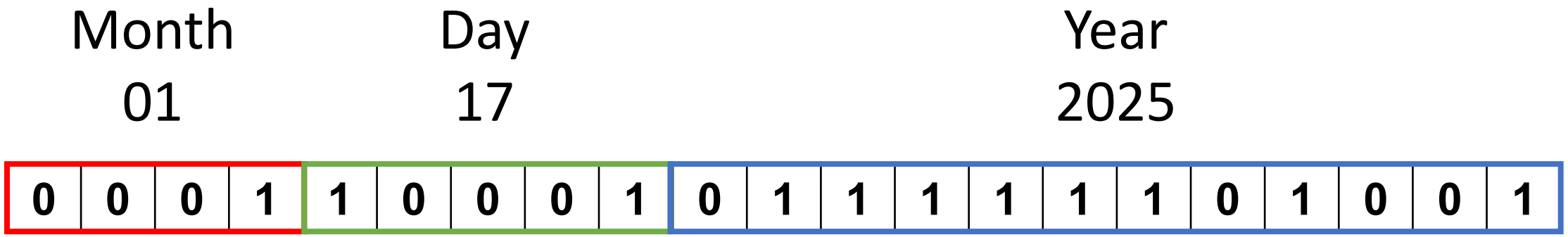
Discussion: Representing Dates

It can be helpful to think logically about how to represent a value before learning how it's done in practice. Let's do that now.

Discuss: How could we use binary to represent a date (i.e., 01/17/2025)?

Representing Dates – Simple Approach

Use 4 bits to represent the month (1-12), 5 bits to represent the day (1-31) and 12 bits to represent the year (1-4095). Convert the month, day, year normally from decimal to binary.



Representing Dates – Actual Approach

In the commonly used Unix Timestamp approach, you count the seconds from a certain date (00:00:00 of 01/01/1970) and convert the number of seconds to binary. Any dates that occur before this time would be negative numbers, and any dates after would be positive numbers!

We use 32 bits to represent each date; the first bit is used to indicate if the number was positive (0) or negative (1), and the remaining 31 bits are used to represent the number of seconds elapsed. Thus, we restrict the number of bits to represent the date to 31 bits.

01/17/2025 2pm -> 1737140400 seconds

0	**1**	**1**	**0**	**0**	**1**	**1**	**1**	**1**	**0**	**0**	**0**	**1**	**0**	**1**	**0**
1	**0**	**1**	**0**	**1**	**0**	**0**	**0**	**1**	**0**	**1**	**1**	**0**	**0**	**0**	**0**

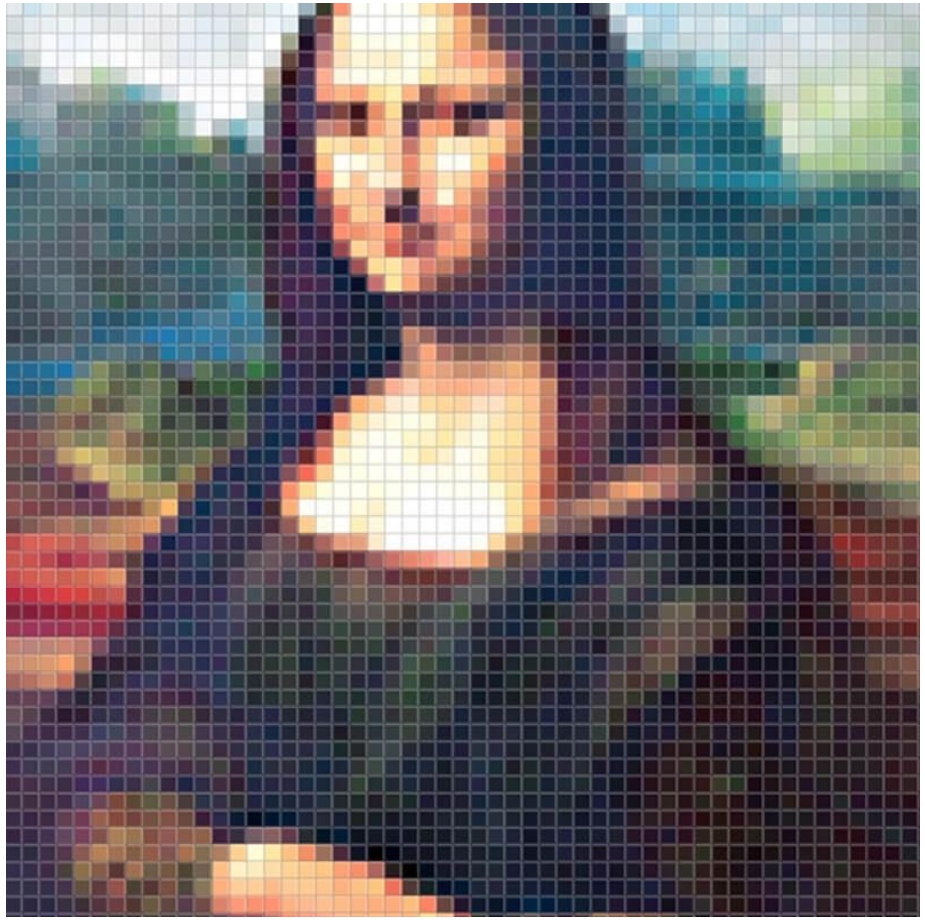
More on dates: https://busyintelligence.blog/2019/05/22/how-does-a-computer-store-data-dates/

Represent Images as Grids of Colors

What if we want to represent an image? How can we convert that to numbers?

First, break the image down into a grid of colors, where each square of color has a distinct hue. A square of color in this context is called a **pixel**.

If we can represent a pixel in binary, we can interpret a series of pixels as an image.



Representing Colors in Binary

We need to represent a single color (a pixel) as a number.

There are a few ways to do this, but we'll focus on **RGB**. Any color can be represented as a combination of Red, Green, and Blue.

Red, green, and blue intensity can be represented using one **byte** each, where 00000000 (0) is none and 11111111 (255) is very intense. Each pixel will therefore require 3 bytes to encode.

Try it out here:
https://www.w3schools.com/colors/colors_rgb.asp

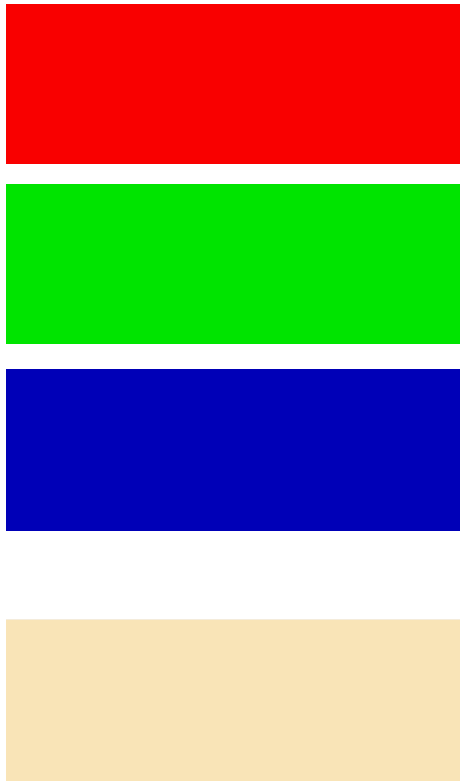
Example: Representing Beige

To make the campus-building beige, we'd need:

Red = 249 = 11111001

Green = 228 = 11100100

Blue = 183 = 10110111



Which makes beige!

Represent Text as Individual Characters

Finally, how do we represent text?

First, we break it down into smaller parts, like with images. In this case, we can break text down into individual **characters**.

For example, the text "Hello World" becomes

H	e	l	l	o	space	W	o	r	l	d

Use a Lookup Table to Convert Characters

Unlike colors, characters don't have a natural connection to numbers.

Instead, we can use a **lookup table** that maps each possible character to an integer. This is how assembly works too!

As long as every computer uses the same lookup table, computers can always translate a set of numbers into the same set of characters.

ASCII is a Simple Lookup Table

For a small set of characters, we can use the encoding system called ASCII. This maps the numbers 0 to 255 to characters. Therefore, one character is represented by one byte.

Check it out here: https://www.asciitable.com/

Dec	Hex	Oct	Chr	Dec	Hex	Oct	HTML	Chr	Dec	Hex	Oct	HTML	Chr	Dec	Hex	Oct	HTML	Chr
0	0	000	NULL	32	20	040	&#032;	Space	64	40	100	&#064;	@	96	60	140	&#096;	`
1	1	001	Start of Header	33	21	041	&#033;	!	65	41	101	&#065;	A	97	61	141	&#097;	a
2	2	002	Start of Text	34	22	042	&#034;	"	66	42	102	&#066;	B	98	62	142	&#098;	b
3	3	003	End of Text	35	23	043	&#035;	#	67	43	103	&#067;	C	99	63	143	&#099;	c
4	4	004	End of Transmission	36	24	044	&#036;	$	68	44	104	&#068;	D	100	64	144	&#100;	d
5	5	005	Enquiry	37	25	045	&#037;	%	69	45	105	&#069;	E	101	65	145	&#101;	e
6	6	006	Acknowledgment	38	26	046	&#038;	&	70	46	106	&#070;	F	102	66	146	&#102;	f
7	7	007	Bell	39	27	047	&#039;	'	71	47	107	&#071;	G	103	67	147	&#103;	g
8	8	010	Backspace	40	28	050	&#040;	(	72	48	110	&#072;	H	104	68	150	&#104;	h
9	9	011	Horizontal Tab	41	29	051	&#041;	)	73	49	111	&#073;	I	105	69	151	&#105;	i
10	A	012	Line feed	42	2A	052	&#042;	*	74	4A	112	&#074;	J	106	6A	152	&#106;	j
11	B	013	Vertical Tab	43	2B	053	&#043;	+	75	4B	113	&#075;	K	107	6B	153	&#107;	k
12	C	014	Form feed	44	2C	054	&#044;	,	76	4C	114	&#076;	L	108	6C	154	&#108;	l
13	D	015	Carriage return	45	2D	055	&#045;	-	77	4D	115	&#077;	M	109	6D	155	&#109;	m
14	E	016	Shift Out	46	2E	056	&#046;	.	78	4E	116	&#078;	N	110	6E	156	&#110;	n
15	F	017	Shift In	47	2F	057	&#047;	/	79	4F	117	&#079;	O	111	6F	157	&#111;	o
16	10	020	Data Link Escape	48	30	060	&#048;	0	80	50	120	&#080;	P	112	70	160	&#112;	p
17	11	021	Device Control 1	49	31	061	&#049;	1	81	51	121	&#081;	Q	113	71	161	&#113;	q
18	12	022	Device Control 2	50	32	062	&#050;	2	82	52	122	&#082;	R	114	72	162	&#114;	r
19	13	023	Device Control 3	51	33	063	&#051;	3	83	53	123	&#083;	S	115	73	163	&#115;	s
20	14	024	Device Control 4	52	34	064	&#052;	4	84	54	124	&#084;	T	116	74	164	&#116;	t
21	15	025	Negative Ack.	53	35	065	&#053;	5	85	55	125	&#085;	U	117	75	165	&#117;	u
22	16	026	Synchronous idle	54	36	066	&#054;	6	86	56	126	&#086;	V	118	76	166	&#118;	v
23	17	027	End of Trans. Block	55	37	067	&#055;	7	87	57	127	&#087;	W	119	77	167	&#119;	w
24	18	030	Cancel	56	38	070	&#056;	8	88	58	130	&#088;	X	120	78	170	&#120;	x
25	19	031	End of Medium	57	39	071	&#057;	9	89	59	131	&#089;	Y	121	79	171	&#121;	y
26	1A	032	Substitute	58	3A	072	&#058;	:	90	5A	132	&#090;	Z	122	7A	172	&#122;	z
27	1B	033	Escape	59	3B	073	&#059;	;	91	5B	133	&#091;	[	123	7B	173	&#123;	{
28	1C	034	File Separator	60	3C	074	&#060;	<	92	5C	134	&#092;	\	124	7C	174	&#124;	|
29	1D	035	Group Separator	61	3D	075	&#061;	=	93	5D	135	&#093;	]	125	7D	175	&#125;	}
30	1E	036	Record Separator	62	3E	076	&#062;	>	94	5E	136	&#094;	^	126	7E	176	&#126;	~
31	1F	037	Unit Separator	63	3F	077	&#063;	?	95	5F	137	&#095;	_	127	7F	177	&#127;	Del

asciichars.com

Translating Text to Numbers

"Yay" ->

Y	a	y

->

89 97 121 ->

01011001 01100001
01111001

Dec	Hex	Oct	Chr	Dec	Hex	Oct	HTML	Chr	Dec	Hex	Oct	HTML	Chr	Dec	Hex	Oct	HTML	Chr
0	0	000	NULL	32	20	040		Space	64	40	100	@	@	96	60	140	`	`
1	1	001	Start of Header	33	21	041	!	!	65	41	101	A	A	97	61	141	a	a
2	2	002	Start of Text	34	22	042	"	"	66	42	102	B	B	98	62	142	b	b
3	3	003	End of Text	35	23	043	#	#	67	43	103	C	C	99	63	143	c	c
4	4	004	End of Transmission	36	24	044	$	$	68	44	104	D	D	100	64	144	d	d
5	5	005	Enquiry	37	25	045	%	%	69	45	105	E	E	101	65	145	e	e
6	6	006	Acknowledgment	38	26	046	&	&	70	46	106	F	F	102	66	146	f	f
7	7	007	Bell	39	27	047	'	'	71	47	107	G	G	103	67	147	g	g
8	8	010	Backspace	40	28	050	(	(	72	48	110	H	H	104	68	150	h	h
9	9	011	Horizontal Tab	41	29	051	)	)	73	49	111	I	I	105	69	151	i	i
10	A	012	Line feed	42	2A	052	*	*	74	4A	112	J	J	106	6A	152	j	j
11	B	013	Vertical Tab	43	2B	053	+	+	75	4B	113	K	K	107	6B	153	k	k
12	C	014	Form feed	44	2C	054	,	,	76	4C	114	L	L	108	6C	154	l	l
13	D	015	Carriage return	45	2D	055	-	-	77	4D	115	M	M	109	6D	155	m	m
14	E	016	Shift Out	46	2E	056	.	.	78	4E	116	N	N	110	6E	156	n	n
15	F	017	Shift In	47	2F	057	/	/	79	4F	117	O	O	111	6F	157	o	o
16	10	020	Data Link Escape	48	30	060	0	0	80	50	120	P	P	112	70	160	p	p
17	11	021	Device Control 1	49	31	061	1	1	81	51	121	Q	Q	113	71	161	q	q
18	12	022	Device Control 2	50	32	062	2	2	82	52	122	R	R	114	72	162	r	r
19	13	023	Device Control 3	51	33	063	3	3	83	53	123	S	S	115	73	163	s	s
20	14	024	Device Control 4	52	34	064	4	4	84	54	124	T	T	116	74	164	t	t
21	15	025	Negative Ack.	53	35	065	5	5	85	55	125	U	U	117	75	165	u	u
22	16	026	Synchronous idle	54	36	066	6	6	86	56	126	V	V	118	76	166	v	v
23	17	027	End of Trans. Block	55	37	067	7	7	87	57	127	W	W	119	77	167	w	w
24	18	030	Cancel	56	38	070	8	8	88	58	130	X	X	120	78	170	x	x
25	19	031	End of Medium	57	39	071	9	9	89	59	131	Y	Y	121	79	171	y	y
26	1A	032	Substitute	58	3A	072	:	:	90	5A	132	Z	Z	122	7A	172	z	z
27	1B	033	Escape	59	3B	073	;	;	91	5B	133	[	[	123	7B	173	{	{
28	1C	034	File Separator	60	3C	074	<	<	92	5C	134	\	\	124	7C	174			|
29	1D	035	Group Separator	61	3D	075	=	=	93	5D	135	]	]	125	7D	175	}	}
30	1E	036	Record Separator	62	3E	076	>	>	94	5E	136	^	^	126	7E	176	~	~
31	1F	037	Unit Separator	63	3F	077	?	?	95	5F	137	_	_	127	7F	177	&#127;	Del

asciichars.com

For More Characters, Use Unicode

There are many characters that aren't available in ASCII (characters from non-English languages, advanced symbols, emoji...) due to the limited size.

The Unicode system represents every character that can be typed into a computer. It uses up to 5 bytes, which can represent up to 1 trillion characters! Find all the Unicode characters here: www.unicode-table.com

The Unicode system is also **actively under development**. The Unicode Consortium regularly updates the standard to add new types of characters and emoji.

Discuss: what are the potential repercussions of using a single standard for all text on computers?

Learning Objectives

- Use **abstraction** to communicate about a system with an appropriate level of detail.

- Use **binary numbers** to represent varying types of data stored on a computer.