CSCI 145 · Introduction to Computing · Fall 2026

Representing Data

Lecture 16 — negative numbers, real numbers, and text

Why is `3 * 0.1 == 0.3` False?

```
>>> 5 * 0.2 == 1
True
>>> 3 * 0.1 == 0.3
False
>>> 3 * 0.1
0.30000000000000004
```

- Python isn't broken
  0.1 can't be stored exactly in bits
- Last time: whole numbers from 0 up, in binary
- Today: what about −5? 0.1? The letter A?
  All of them are still just bits

Outline

- Negative numbers
- Real numbers
- Text

Part 1 of 3

Negative numbers

> A word has 2^{64} patterns
> Give half of them to negative numbers

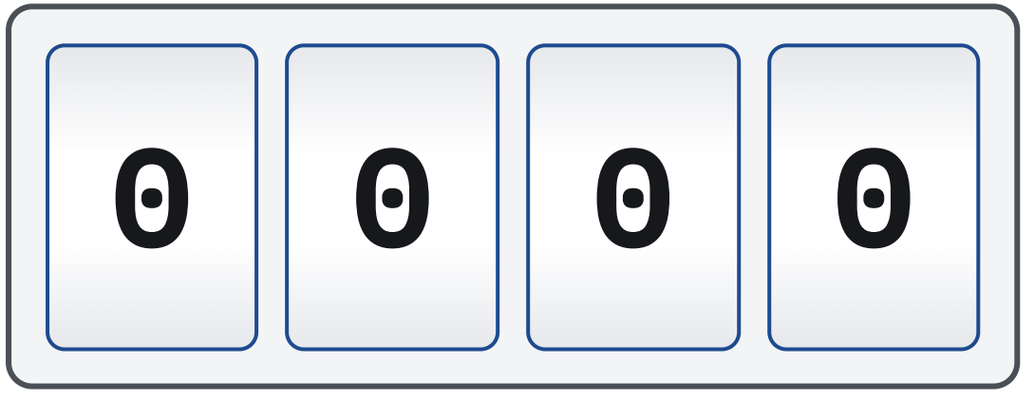
First try: a sign bit

To keep the pictures small, we'll use one byte (8 bits) instead of a 64-bit word



- Use the leftmost bit for the sign: 0 means +, 1 means −
- Problem 1: two different zeros
- Problem 2: adding breaks
  `00000101` + `10000101` gives `10001010`, which would be −10, not 0
- Computers use a smarter code: **two's complement**

Demo: a 4-bit odometer



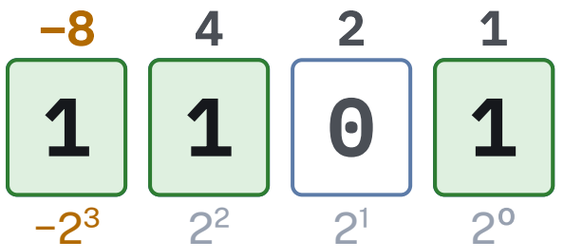
- Each wheel shows 0 or 1
- Count up from `0000`: `0001`, `0010`, `0011`, ...
- What comes after `1111`?
- Start at `0000` and go back one step
  What does it show?

Counting down past zero

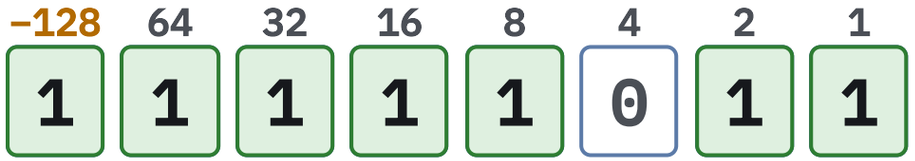


- Back one step from `0000`, every wheel rolls over: `1111` is −1
- Keep going: `1110` is −2, `1101` is −3
- Stop at `1000`: −8
- Half the patterns go to −8 to −1, the other half to 0 to 7
- Every negative number starts with 1

The leftmost place is negative



−8 + 4 + 1 = −3, as on the odometer

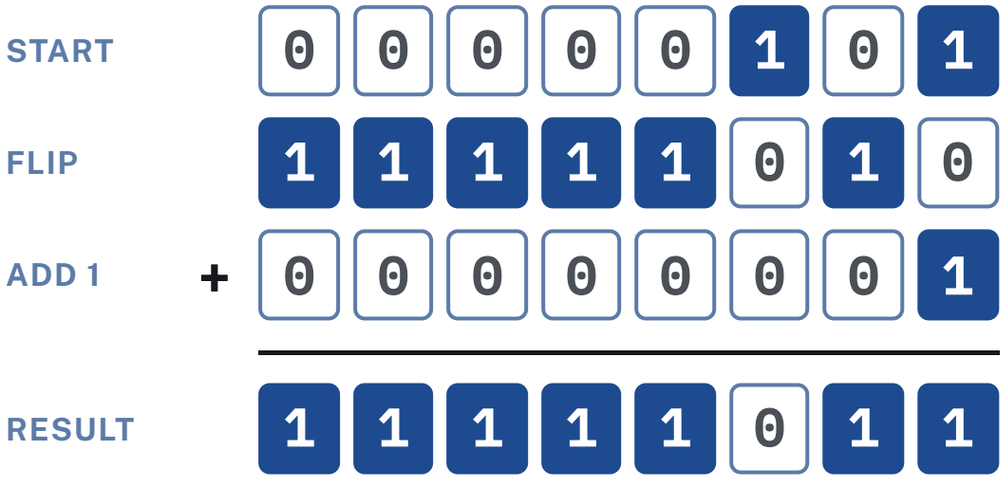


−128 + 64 + 32 + 16 + 8 + 2 + 1 = −5

- Same places as before, except the leftmost one counts as negative
- 8 bits: `10000000` is −128, `01111111` is 127

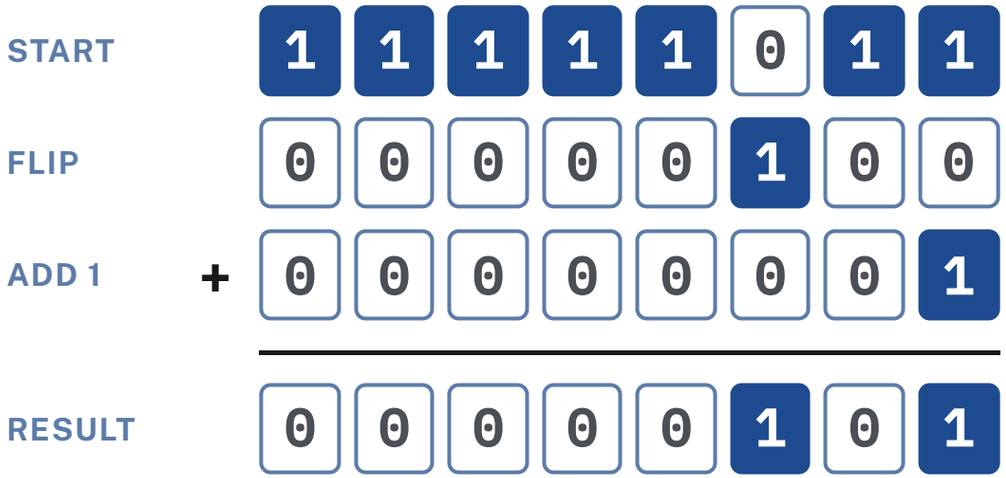
Negating: flip the bits, add 1

5 to −5



11111011 is −5, as on the last slide

−5 back to 5



00000101 is 5 again

- The same two steps work in both directions

How far can we count?

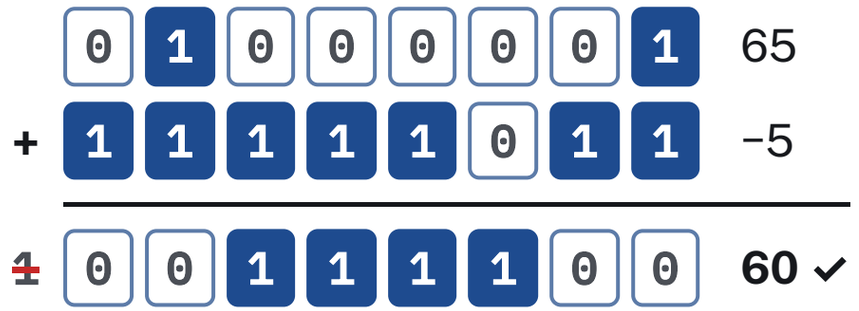
# of Bits	# of Patterns	Smallest	Largest
4	16	−8	7
8	256	−128	127
16	65,536	−32,768	32,767
32	about 4 billion	−2,147,483,648	2,147,483,647
64	about 18 quintillion	−9,223,372,036,854,775,808	9,223,372,036,854,775,807

- `n` bits: -2^{n-1} to $2^{n-1} - 1$
- One more negative number than positive: 0 uses up a pattern on the positive side
- Python ints grow as big as they need to, so you won't hit these limits
  Most other languages use fixed sizes

Adding just works, until the bits run out

Illustrating with 8 bits:

65 + (−5)



Drop the carry out of the left end

65 + 65



Overflow: 130 doesn't fit in 8 bits

- Same adding as last time: no special rules for negative numbers
- Modern computers usually use 64 bits for an integer: up to about 9 quintillion
- Overflow happens when a value is stored in fewer bits than it needs
- In 2014, Gangnam Style passed 2,147,483,647 views, the 32-bit limit
  YouTube had already switched its view counter to 64 bits

Eight 1s

What number is **11111111** in 8-bit two's complement?



−128 + 64 + 32 + 16 + 8 + 4 + 2 + 1

= −128 + 127 = −1

Check: flip to `00000000`, add 1: `00000001` is 1

A 255

B −1

C −127

D −128

Answer: B

One step back from 0, like the odometer

Why not the others?

A: 255 reads it without a negative place

C: −127 reads it with a sign bit

D: −128 is `10000000`

Get set up

1. Download day16.zip
2. Drag `day16.zip` to your `class` folder and extract it there

   Mac — double-click it. **Windows** — right-click it and choose "Extract All..."

   Once you see the `day16` folder, delete `day16.zip`
3. Launch Thonny

   We'll use its shell to check our work today

Activity A

On the worksheet, do problems 1–3 by hand

PATTERNS

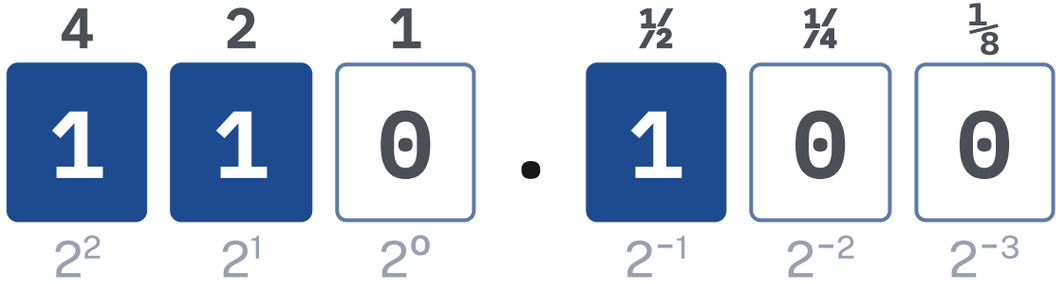
Leftmost place	worth −128 (8 bits)
Negate a number	flip every bit, then add 1
Add	as before; drop a carry out of the left end

Part 2 of 3

Real numbers

> Python stores real numbers as **floats**
>
> You won't convert floats to bits by hand
>
> What matters: floats are approximate

Places after the point



- Each place is half the place to its left, past the point too
- Same idea as decimals: 0.75 is 7 tenths and 5 hundredths

$0.1_2 = ½ = 0.5$

$0.01_2 = ¼ = 0.25$

$0.11_2 = ½ + ¼ = 0.75$

$110.1_2 = 4 + 2 + ½ = 6.5$

Fractions to binary: multiply by 2

NOT TESTED, JUST FYI

Write 0.625 in binary

Fraction	× 2	Bit
0.625	1.25	**1**
0.25	0.5	**0**
0.5	1	**1**



- Double the fraction:
  the whole part, 0 or 1, is the next bit
- Keep only the part after the point and repeat
- Stop when the fraction is 0
- Read the bits from the top down:
  0.625 is 0.101_2 (½ + ⅛)

Now try 0.1

NOT TESTED, JUST FYI

Fraction	× 2	Bit
0.1	0.2	0
0.2	0.4	0
0.4	0.8	0
0.8	1.6	1
0.6	1.2	1
0.2	0.4	0
⋮	⋮	⋮

- Same steps as for 0.625
- 0.2 again! From here the same rows repeat forever
- 0.1 is $0.0001100110011\ldots_2$
- A float has only 64 bits,
  so the bits get cut off and rounded

Like ⅓ in decimal

	Base 10, 4 digits	Base 2, 64 bits
Can't write exactly	⅓ = 0.3333…	0.1 = `0.000110011…`$_2$
Stored as	0.3333	0.1000000000000000055511…
Times 3	0.9999	0.3000000000000004
Equal?	`0.9999 == 1` False	`3 * 0.1 == 0.3` False

- Some fractions never end: ⅓ in base 10, $\frac{1}{10}$ in base 2
- Sums of halves, quarters, eighths, … are exact: 0.5, 0.25, 0.75

Floating point

NOT TESTED, JUST FYI

Scientific notation: $6{,}500 = 6.5 \times 10^3$

In binary: $6.5 = 110.1_2 = 1.101_2 \times 2^2$



- A 64-bit float stores a sign, an exponent, and a fraction
- The point “floats”: the exponent says where it goes
- 52 bits of fraction give about 16 decimal digits
- Every computer follows the same standard, IEEE 754
  The details are for CS 202

Floats in Python

```
>>> 0.1 + 0.2
0.30000000000000004
>>> 0.5 + 0.25 == 0.75
True
>>> round(0.1 + 0.2, 2)
0.3
>>> abs((0.1 + 0.2) - 0.3) < 0.000001
True
```

- Try these in the Thonny shell
- Halves and quarters are exact
- Round before you show a float
- Don't compare floats with `==`
  Check that they're close
- Ints are exact, floats are not
- `5 * 0.2 == 1` was luck:
  the rounding errors cancelled out

What you need to know about floats

Can a float hold every value between 0 and 1?

No: there are gaps, and 0.1 falls in one

Why test `f <= 0` instead of `f == 0`?

Rounding can land `f` just past 0, never on it

`0.3 - 0.1 - 0.1 - 0.1` is `-2.8e-17`, not 0

Can a 64-bit float hold numbers bigger than 2^{64}?

Yes: the exponent moves the point

The cost: big floats have big gaps

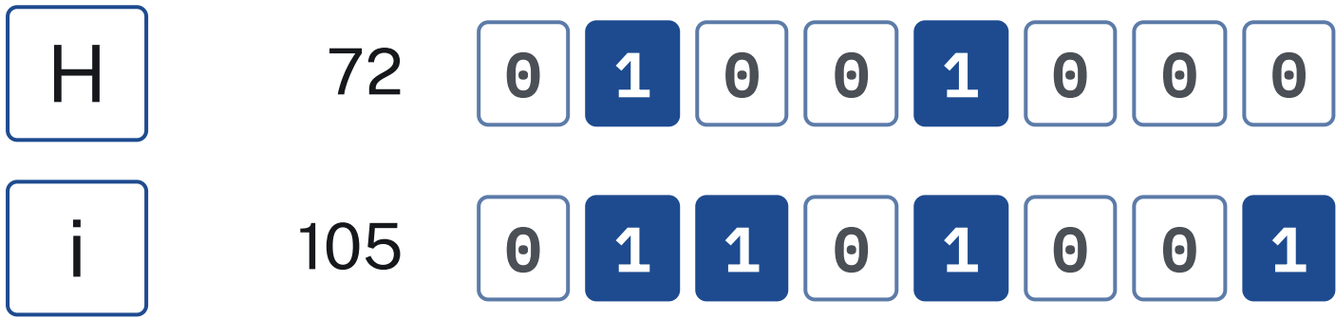
(For scale only: the largest float is about 1.8×10^{308}, while 2^{64} is about 1.8×10^{19})

Part 3 of 3

Text

> Each character is stored as a number, its **code**
> Everyone has to agree on the same codes

Text is numbers too



- Give every character a number, its code
- Store each code in binary: one byte per character
- Everyone has to use the same table, so text reads the same on every computer
- A text file (Lecture 14) is just these bytes, one after another

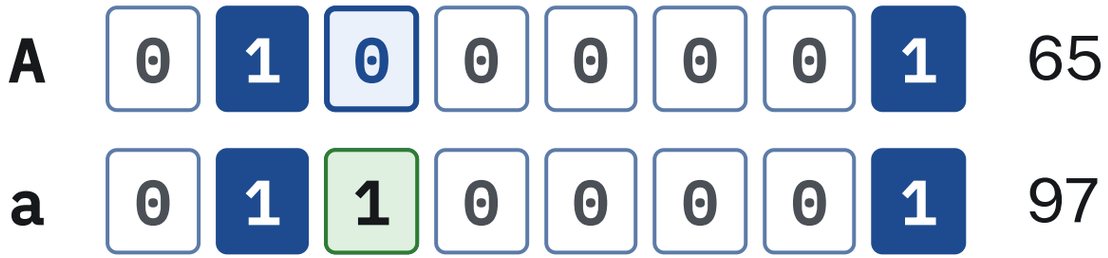
The ASCII table

Row: first hex digit · Column: second hex digit · Codes 32 to 127 shown · Full ASCII table

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
2	space	!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	del

- The standard code for English since the 1960s
- **A** is in row 4, column 1:
  41 hex = 65
- **a** is 61 hex = 97
- The digit **0** is 30 hex = 48, not 0
- Space is 20 hex = 32
- Codes 0 to 31 are invisible: newline is 10

Patterns in ASCII



- Uppercase and lowercase are 32 apart
- They differ in one bit, the 32s place
- Letters are in order: **A** is 65, **B** is 66, ..., **Z** is 90;
  a is 97, **b** is 98, ..., **z** is 122
- So `"Zebra" < "apple"` is True: strings compare by code
- ASCII uses 7 bits: 128 codes, English only

Beyond English: Unicode

Unicode gives a code to every character in every language, plus emoji: over 150,000 of them

Character	Code	Bytes	UTF-8 bytes (hex)	First byte (binary)
A	65	1	`41`	`01000001`
é	233	2	`C3 A9`	`11000011`
中	20,013	3	`E4 B8 AD`	`11100100`
	128,568	4	`F0 9F 98 B8`	`11110000`

- The first 128 codes match ASCII
- **UTF-8** stores each code in 1 to 4 bytes: most web pages and text files use it

How UTF-8 picks the number of bytes NOT TESTED, JUST FYI

By the size of the code: under 128, 1 byte; under 2,048, 2; under 65,536, 3; the rest, 4

The first byte's leading bits tell a reader how many bytes to take:
`0` for 1, `110` for 2, `1110` for 3, `11110` for 4

Codes in Python: `ord` and `chr`

```
>>> ord("A")
65
>>> chr(65)
'A'
>>> chr(ord("A") + 1)
'B'
>>> ord("é")
233
>>> chr(128568)
'😸'
```

- `ord(ch)` gives the code of one character
- `chr(code)` gives the character with that code
- Codes are ints, so you can do arithmetic: one letter later is `chr(ord(ch) + 1)`
- Python strings are Unicode

Same bits, different meanings

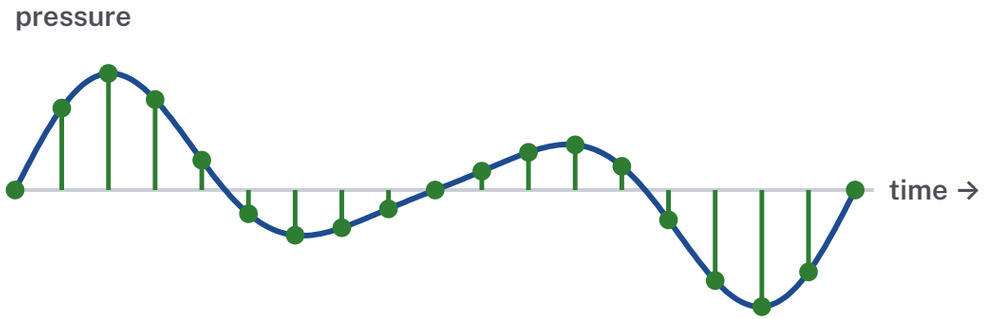
Data	Standard
Whole numbers	binary, two's complement
Real numbers	IEEE 754 floats
Text	ASCII, Unicode, UTF-8
Colors	RGB, one byte each
Images	PNG, JPEG (Lecture 17)
Sound	WAV, MP3
Video	MP4



- As a whole number: 65
- As text: **A**
- The bits don't say what they mean
  The program has to know the standard
- Every file format is an agreement about bits

Sound is numbers too

OPTIONAL



- Sound is air pressure going up and down
- **Sampling**: measure the pressure at regular times
- CD sound: 44,100 samples a second, each a 16-bit two's complement int, −32,768 to 32,767
- Louder sound, bigger numbers

Activity B

On the worksheet, do problems 4–6 with the ASCII table

Then check your answers in the Thonny shell

PATTERNS

Character to code	`ord("G")`
Code to character	`chr(71)`
Hex to decimal	`int("4F", 16)`
Decimal to hex	`hex(71)`

Activity C

In Thonny, open `cs145/class/day16/day16_text.py`
Write `close_enough`, `to_codes`, and `from_codes`
If you have time, write `shift_word`

PATTERNS

Distance between two numbers	`abs(a - b)`
Code of a character	`ord(ch)`
Character from a code	`chr(code)`
Build a list	`result.append(code)`
Is it a lowercase letter?	`ch.islower()`
Wrap around after z	`% 26`

To do

- Review this slide deck and Notes 13
- Exam 3 (Lectures 10–14) is Friday 11/6 in lab
- Quiz 8 (Lectures 15–16) is at the start of next class
- Homework 5 is due Thursday 11/12

} See practice materials

Next time

Lecture 17 — images

A photo is a grid of colors, three bytes each