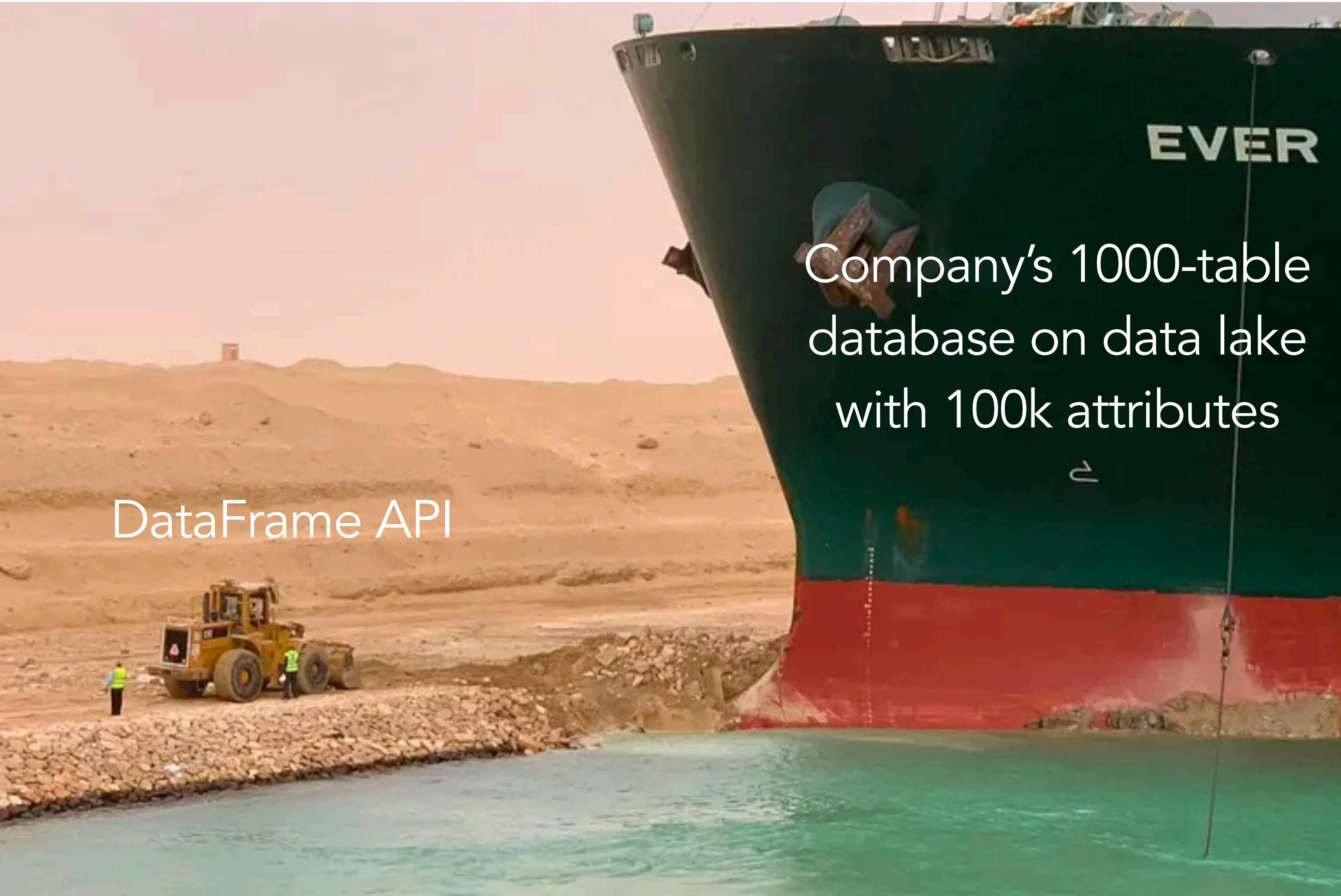


DSC 204a

Scalable Data Systems

- Haojian Jin



Components and Grading

- 3 Programming Assignments: **40%** ($8\% + 16\% + 16\%$)
 - No late days! Plan your work well ahead.
- Final Exam (06/14/2023 3pm-6pm): **40%** ?
- Peer Instruction Activities: **20%**
- Extra Credit Peer Evaluation Activities: **4%** (likely)

Logistics

- Final Exam (06/14/2023 3pm-6pm): **40%**
- Comprehensive exam requirement: P/N =>
- The programming assignments are projects.

Grading Scheme (grade is the better of the two)

| Grade | Absolute Cutoff (\geq) | Relative Bin (Use strictest) |
|-------|----------------------------|------------------------------|
| A+ | 95 | Highest 5% |
| A | 90 | Next 10% (5-15) |
| A- | 85 | Next 15% (15-30) |
| B+ | 80 | Next 15% (30-45) |
| B | 75 | Next 15% (45-60) |
| B- | 70 | Next 15% (60-75) |
| C+ | 65 | Next 5% (75-80) |
| C | 60 | Next 5% (80-85) |
| C- | 55 | Next 5% (85-90) |
| D | 50 | Next 5% (90-95) |
| F | < 50 | Lowest 5% |

Website updating

| EVENT | DATE | DESCRIPTION | COURSE MATERIAL |
|---------|-------------------------|--|---|
| Lecture | 04/03/2023 Monday | Introduction & logistics [slides] | |
| Lecture | 04/05/2023 Wednesday | Basics of Machine Resources | Suggested Readings: <ul style="list-style-type: none">◦ Ch. 1, 2.1-2.3, 2.12, 4.1, and 5.1-5.5 of CompOrg Book |

Where we are in the class?

Foundations of Data Systems

- **Digital representation of Data** → Basics of Computer Organization → Basics Operating Systems
- Array data representation → Storage Memory Hierarchy → Cache
- Data encoding and evolution

Scaling Distributed Systems

Data Processing and Programming model

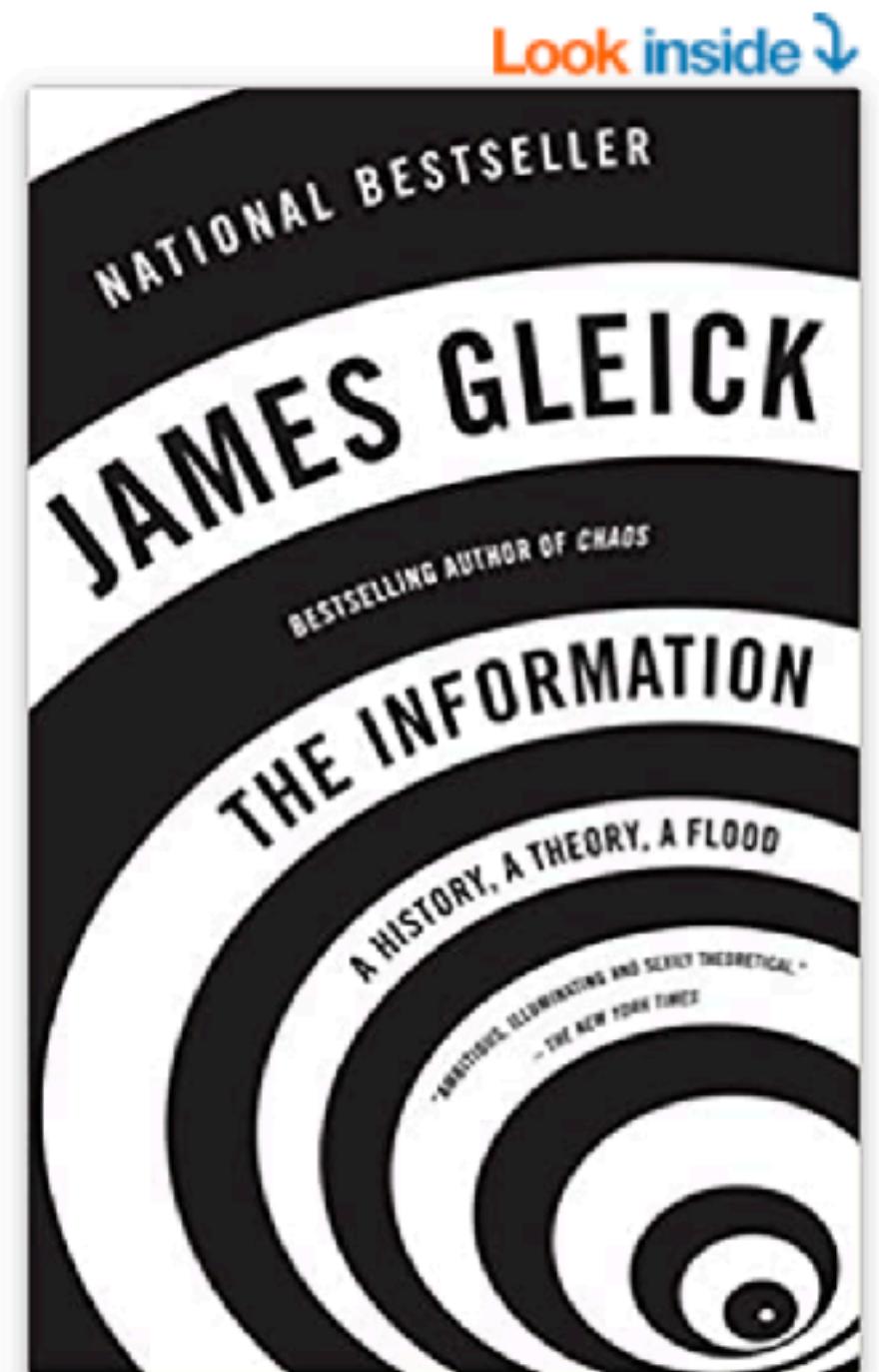
Digital Representation of Data

- **Bits:** 01010101
- **Data type:** Boolean, Byte, Integer, “floating point” number (Float), Character, and String
- **Data structure:** Array, Linked list, Tuple, Graph, etc.

Programmer

Machine?

Why bits?



Listen



See this image

The Information: A History, A Theory, A Flood Paperback – Illustrated, March 6, 2012

by [James Gleick](#) (Author)

1,339 ratings

Editors' pick Best Science Fiction & Fantasy

Searching •••

S +

[See all formats and editions](#)

Kindle

\$9.99

[Read with Our Free App](#)

Audiobook

\$0.00

[Free with your Audible trial](#)

Hardcover

\$13.25 - \$39.94

[Other new, used and collectible from \\$2.39](#)

Paperback

\$12.23

[Other new and used from \\$2.45](#)

Audio CD

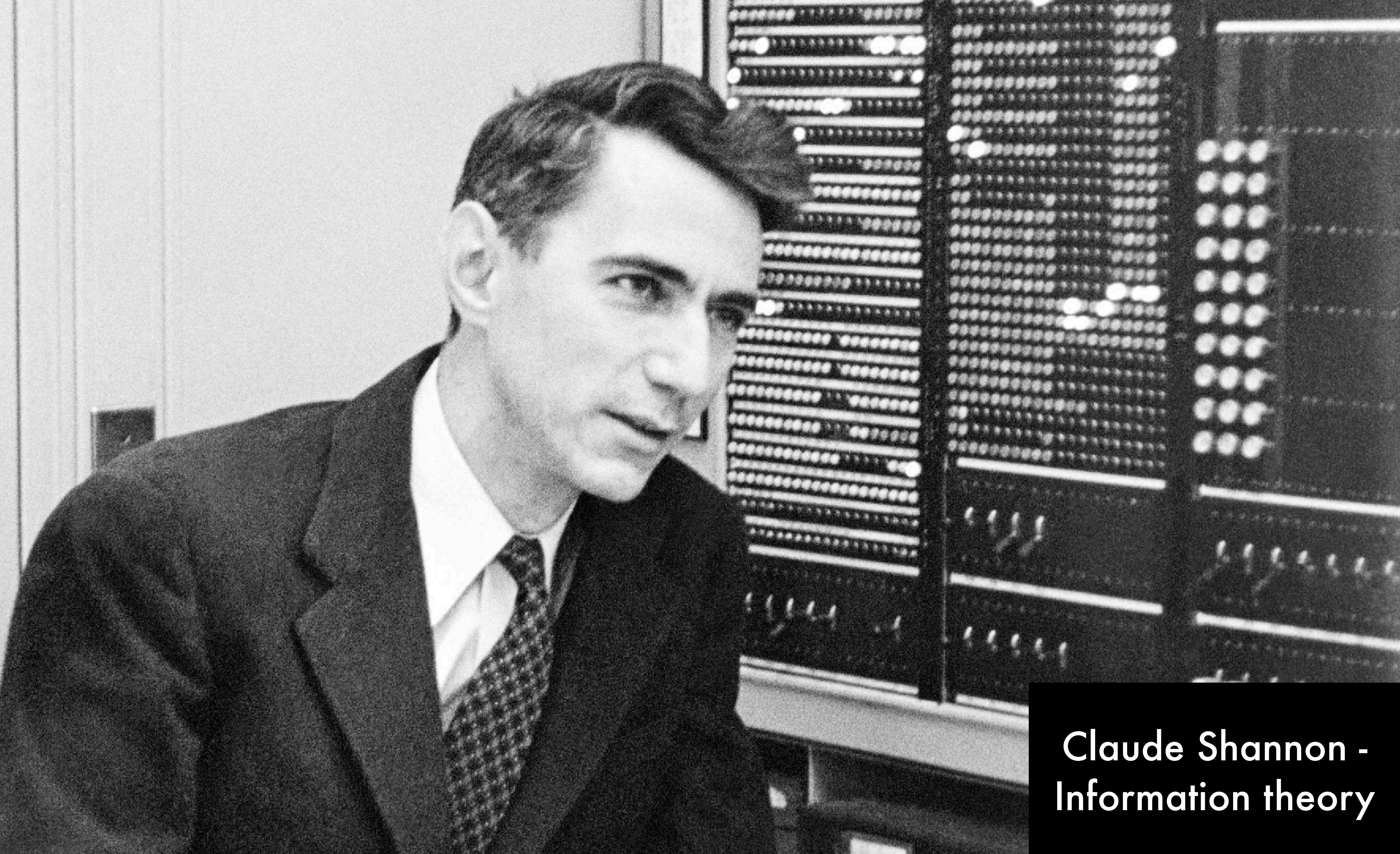
\$25.04

[8 Used from \\$21.06](#)

From the bestselling author of the acclaimed *Chaos* and *Genius* comes a thoughtful and provocative exploration of the big ideas of the modern era: Information, communication, and information theory.

Acclaimed science writer James Gleick presents an eye-opening vision of how our relationship to information has transformed the very nature of human consciousness. A fascinating intellectual journey through the history of communication and information, from the language of Africa's talking drums to the invention of written alphabets; from the electronic transmission of code to the

[▼ Read more](#)



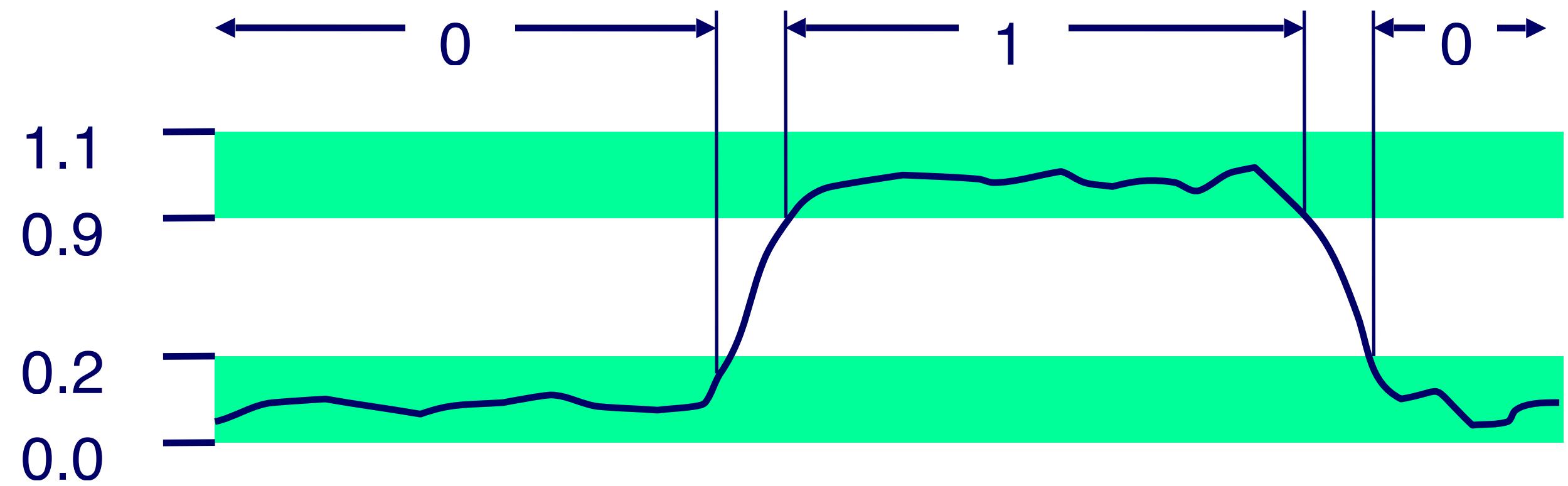
Claude Shannon -
Information theory

Before bits

- By 1948 more than 125 million conversations passed daily through the Bell System's 138 million miles of cable and 31 million telephone sets.
- Count words, characters, electricity consumption, minutes.

Why bits?

- Physics! Electronic Implementation
 - Easy to store with bistable elements.
 - e.g., high-low/off-on electromagnetism on disk.
 - Reliably transmitted on noisy and inaccurate wires
- Very expressive and efficient.
- Quantum computers?



Count everything in binary

- **Base 2 Number Representation**
 - 0, 1, 10, 11, 100, 101, ...
 - Represent 15213_{10} as 0011 1011 0110 1101₂
 - Represent 1.20_{10} as 1.0011 0011 0011 0011 [0011]...₂
 - Represent $(1.5213 \times 10^4)_{10}$ as $(1.1101\ 1011\ 0110\ 1 \times 2^{13})_2$
- **Represent negative numbers as ...?**
 - (we'll come back to this)

Encoding Byte Values

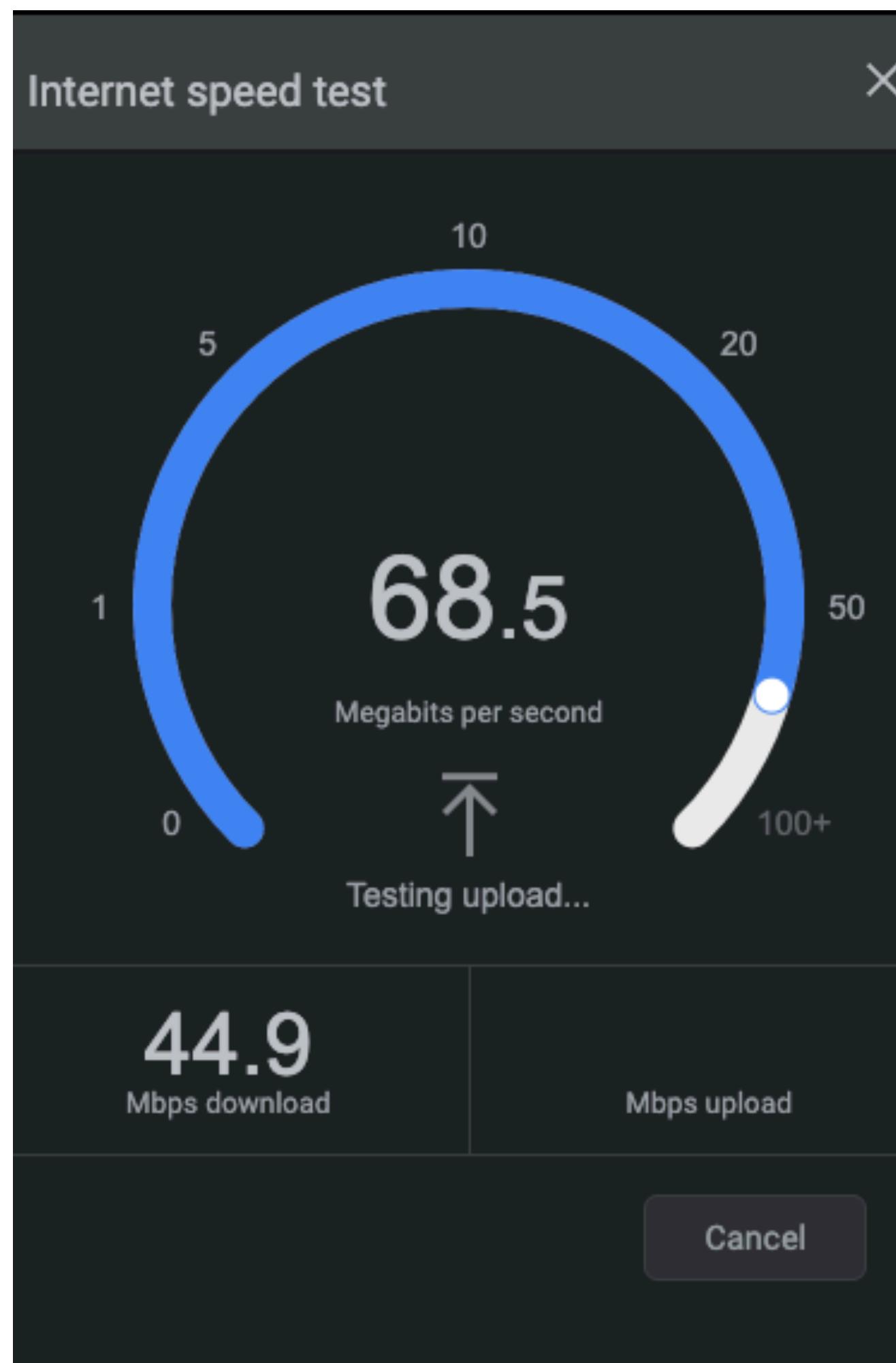
- **Byte = 8 bits. Why?**

- Decimal: 0_{10} to 255_{10}
 - $255 = 2^8 - 1$
- Binary: $0000\ 0000_2$ to $1111\ 1111_2$
- *Hexadecimal*: 00_{16} to FF_{16} .
 - Why? Example: #FF5733, RGB 255, 87, 51
 - Base 16 number representation
 - Use characters '0' to '9' and 'A' to 'F'
 - Write in C with leading '0x', either case

- $0101\ 1010_2 = 0x5a = 0x5A = 0X5a$

15213: 0011 1011 0110 1101
3 B 6 D

| Hex | Decimal | Binary |
|-----|---------|--------|
| 0 | 0 | 0000 |
| 1 | 1 | 0001 |
| 2 | 2 | 0010 |
| 3 | 3 | 0011 |
| 4 | 4 | 0100 |
| 5 | 5 | 0101 |
| 6 | 6 | 0110 |
| 7 | 7 | 0111 |
| 8 | 8 | 1000 |
| 9 | 9 | 1001 |
| A | 10 | 1010 |
| B | 11 | 1011 |
| C | 12 | 1100 |
| D | 13 | 1101 |
| E | 14 | 1110 |
| F | 15 | 1111 |



| Name | Size | Kind |
|---------------------|--------|------------|
| HB50 cupcakes.JPG | 2 MB | JPEG image |
| Roller Skating.JPG | 1.3 MB | JPEG image |
| 50HBJukebox2.jpg | 720 KB | JPEG image |
| Facebook.tiff | 399 KB | TIFF image |
| 7_days_to_enrol.png | 173 KB | PNG image |
| JoggingShoes.jpg | 71 KB | JPEG image |

Digital Representation of Data

- The size and *interpretation* of a data type depends on PL
- A **Byte** (B; 8 bits) is typically the basic unit of data types
 - CPU can't address anything smaller than a byte.
- **Boolean:**
 - Examples in data sci.: Y/N or T/F responses
 - Just 1 bit needed but actual size is almost always 1B, i.e., 7 bits are wasted! (**Q:** Why?)
- **Integer:**
 - Examples in data science: #friends, age, #likes
 - Typically 4 bytes; many variants (short, unsigned, etc.)
 - Java *int* can represent -2^{31} to $(2^{31} - 1)$; C *unsigned int* can represent 0 to $(2^{32} - 1)$;
Python3 *int* is effectively unlimited length (PL magic!)

Digital Representation of Data

- **Bits:** All digital data are sequences of 0 & 1 (binary digits)
- **Data type:** First layer of abstraction to interpret a bit sequence with a human-understandable category of information; interpretation fixed by the PL
 - Example common datatypes: Boolean, Byte, Integer, “floating point” number (Float), Character, and String
- **Data structure:** A second layer of abstraction to organize multiple instances of same or varied data types as a more complex object with specified properties
 - Examples: Array, Linked list, Tuple, Graph, etc.

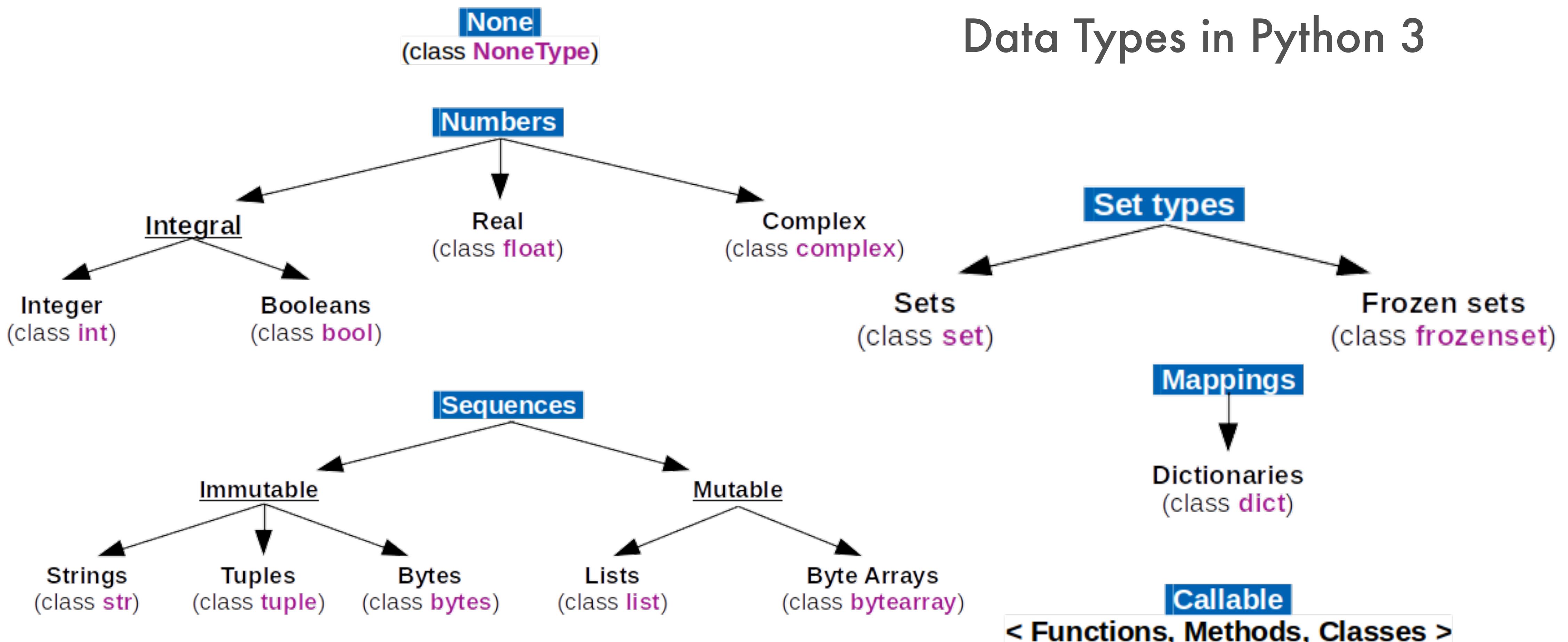
Combine bytes to make *scalar data types*

| C Data Type | Size (# of bytes) | |
|----------------|-------------------|----------------|
| | Typical 32-bit | Typical 64-bit |
| char | 1 | 1 |
| short | 2 | 2 |
| int | 4 | 4 |
| long | 4 | 8 |
| float | 4 | 4 |
| double | 8 | 8 |
| pointer | 4 | 8 |

“ILP32” “LP64”

Estimate the object size in your memory, network, storage!

Digital Representation of Data



Digital Representation of Data

Q: How many unique data items can be represented by 3 bytes?

- Given k bits, we can represent 2^k unique data items
- 3 bytes = 24 bits $\Rightarrow 2^{24}$ items, i.e., 16,777,216 items
- Common approximation: 2^{10} (i.e., 1024) $\sim 10^3$ (i.e., 1000); recall kibibyte (KiB) vs kilobyte (KB) and so on

Q: How many bits are needed to distinguish 97 data items?

- For k unique items, invert the exponent to get $\log_2(k)$
- But #bits is an integer! So, we only need $\lceil \log_2(k) \rceil$
- So, we only need the next higher power of 2
- $97 \rightarrow 128 = 2^7$; so, 7 bits

Digital Representation of Data

Q: How to convert from decimal to binary representation?

- Given decimal n, if power of 2 (say, 2^k), put 1 at bit position k; if $k=0$, stop; else pad with trailing 0s till position 0
- If n is not power of 2, identify the power of 2 just below n (say, 2^k); #bits is then k; put 1 at position k
- Reset n as $n - 2^k$; return to Steps 1-2
- Fill remaining positions in between with 0s

| | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | Position/Exponent of 2 |
|------------|-----|----|----|----|---|---|---|---|------------------------|
| Decimal | 128 | 64 | 32 | 16 | 8 | 4 | 2 | 1 | Power of 2 |
| 5_{10} | | | | | | 1 | 0 | 1 | |
| 47_{10} | | | | 1 | 0 | 1 | 1 | 1 | |
| 163_{10} | 1 | 0 | 1 | 0 | 0 | 0 | 1 | 1 | |
| 16_{10} | | | | 1 | 0 | 0 | 0 | 0 | |

Q: Binary to decimal?

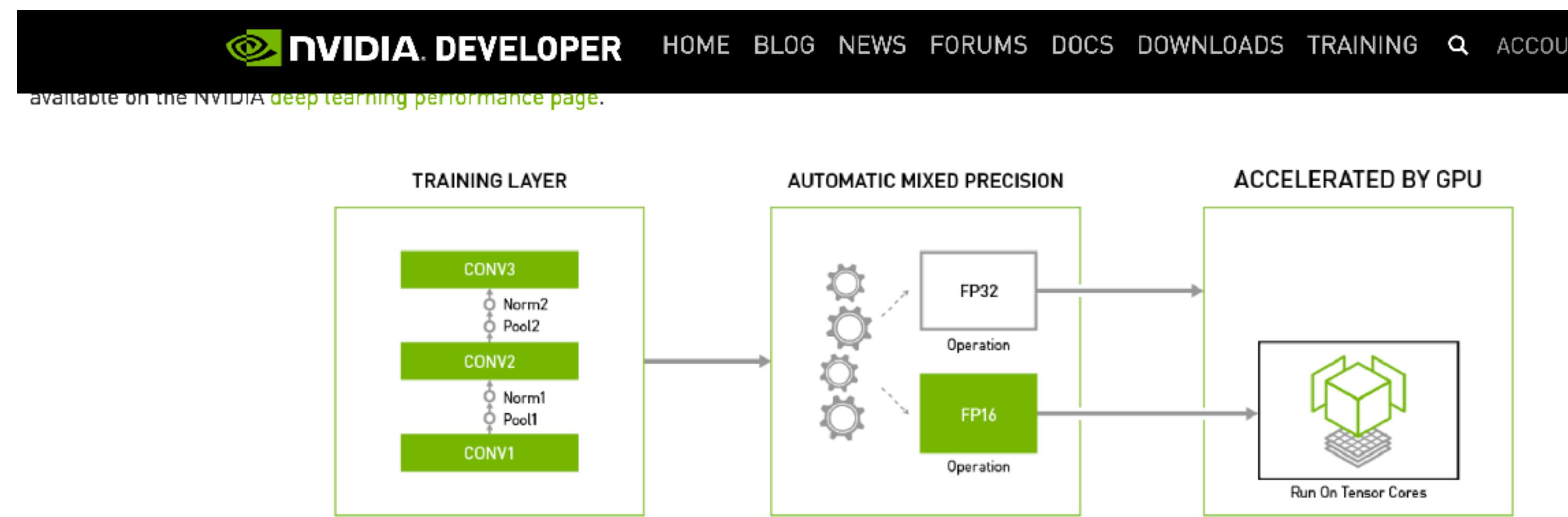
Digital Representation of Data

- *Hexadecimal* representation is a common stand-in for binary representation; more succinct and readable
 - Base 16 instead of base 2 cuts display length by $\sim 4x$
 - Digits are 0, 1, ... 9, A (10_{10}), B, ... F (15_{10})
 - From binary: combine 4 bits at a time from lowest

| Decimal | Binary | Hexadecimal | |
|------------|----------------|-------------|-----------------------|
| 5_{10} | 101_2 | 5_{16} | Alternative notations |
| 47_{10} | $10\ 1111_2$ | $2F_{16}$ | |
| 163_{10} | $1010\ 0011_2$ | $A3_{16}$ | $0xA3$ or $A3_H$ |
| 16_{10} | $1\ 0000_2$ | 10_{16} | |

Digital Representation of Data

- **Float:**
 - Examples in data sci.: salary, scores, model weights
 - IEEE-754 single-precision format is 4B long; double-precision format is 8B long
 - Java and C *float* is single; Python *float* is double!

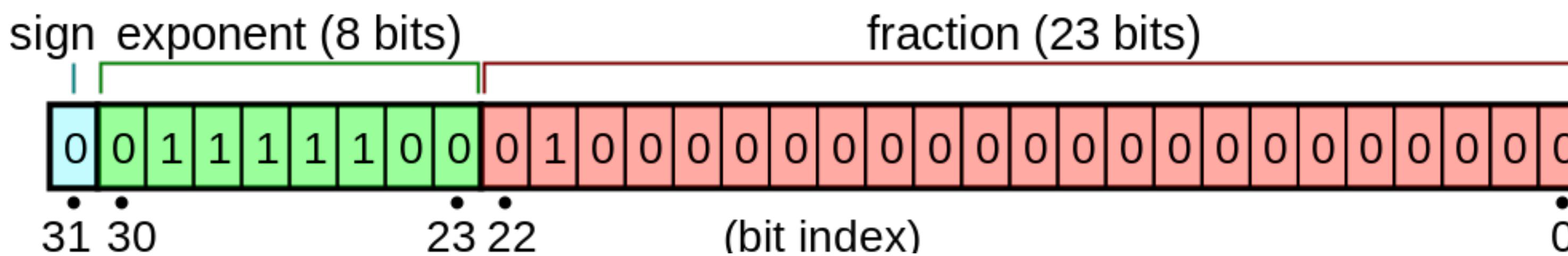


Using Automatic Mixed Precision for Major Deep Learning Frameworks

Digital Representation of Data

- **Float:**

- Standard IEEE format for single (aka binary32):

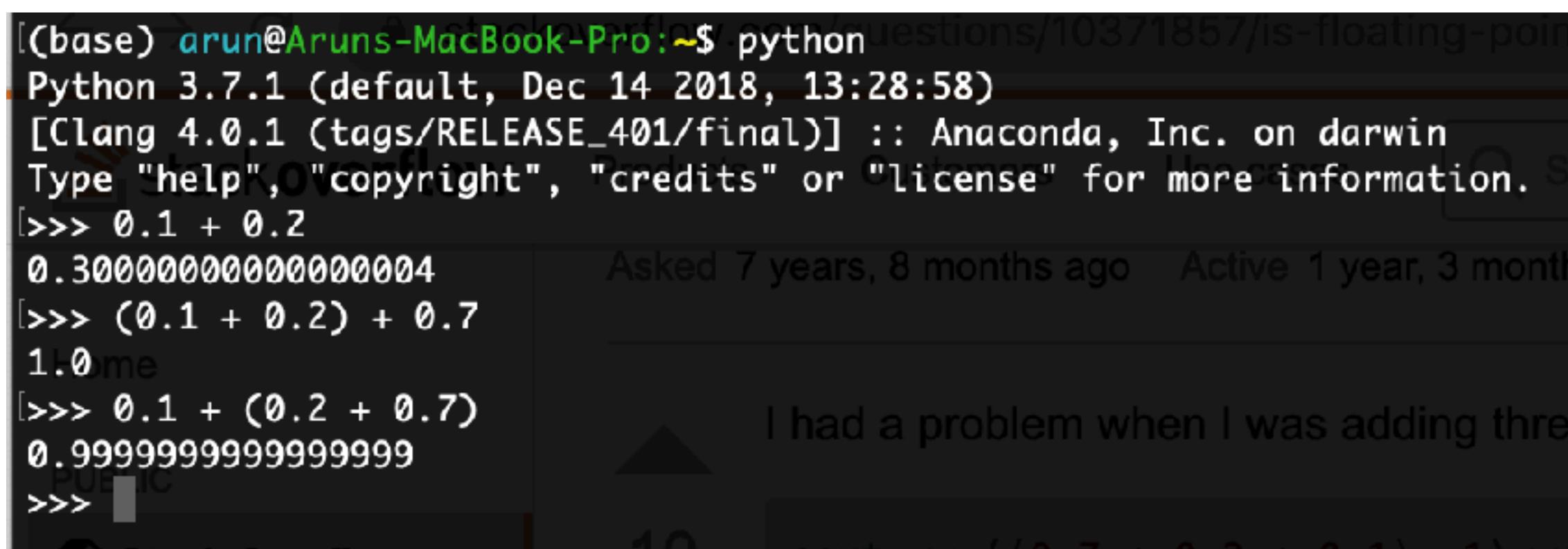


$$(-1)^{sign} \times 2^{exponent-127} \times \left(1 + \sum_{i=1}^{23} b_{23-i} 2^{-i}\right)$$

$$(-1)^0 \times 2^{124-127} \times (1 + 1 \cdot 2^{-2}) = (1/8) \times (1 + (1/4)) = 0.15625$$

Digital Representation of Data

- Due to representation imprecision issues, floating point arithmetic (addition and multiplication) is not associative!



A screenshot of a Stack Overflow question titled "Is floating-point arithmetic associative?". The question was asked 7 years, 8 months ago and has been active for 1 year, 3 months. It has 10 upvotes and 10 downvotes. The question text reads: "I had a problem when I was adding three floating point numbers. I expected the result to be 1.0, but it wasn't. I'm not sure if this is a bug or if floating-point arithmetic is not associative." Below the question, there is a Python terminal session showing non-associativity:

```
(base) arun@Aruns-MacBook-Pro:~$ python
Python 3.7.1 (default, Dec 14 2018, 13:28:58)
[Clang 4.0.1 (tags/RELEASE_401/final)] :: Anaconda, Inc. on darwin
Type "help", "copyright", "credits" or "license" for more information.
[>>> 0.1 + 0.2
0.3000000000000004
[>>> (0.1 + 0.2) + 0.7
1.0
[>>> 0.1 + (0.2 + 0.7)
0.9999999999999999
[>>>
```

- In binary32, special encodings recognized:
 - Exponent 0xFF and fraction 0 is +/- “Infinity”
 - Exponent 0xFF and fraction <> 0 is “NaN”
 - Max is ~ 3.4×10^{38} ; min +ve is ~ 1.4×10^{-45}

Digital Representation of Data

- More float standards: double-precision (float64; 8B) and half-precision (float16; 2B); different #bits for exponent, fraction
- Float16 is now common for deep *learning* parameters:
 - Native support in PyTorch, TensorFlow, etc.; APIs also exist for weight quantization/rounding post training
 - NVIDIA Deep Learning SDK support mixed-precision training; 2-3x speedup with similar accuracy!
 - New processor hardware (FPGAs, ASICs, etc.) enable arbitrary precision, even 1-bit (!), but accuracy is lower

Digital Representation of Data

```
void show_squares()
{
    int x;
    for (x = 5; x <= 5000000; x*=10)
        printf("x = %d x^2 = %d\n", x, x*x);
}
```

x = 5 x² = 25

x = 50 x² = 2500

x = 500 x² = 250000

x = 5000 x² = 25000000

x = 50000 x² = -1794967296

x = 500000 x² = 891896832

x = 5000000 x² = -1004630016

In C, int uses a 32-bit

Digital Representation of Data

```
import sys

i = sys.maxsize
print(i)
# 9223372036854775807
print(i == i + 1)
# False
i += 1
print(i)
# 9223372036854775808

f = sys.float_info.max
print(f)
# 1.7976931348623157e+308
print(f == f + 1)
# True
f += 1
print(f)
# 1.7976931348623157e+308
```

```
import numpy as np

a = np.array([3095693933], dtype=int)
s = np.sum(a)
print(s)
# 3095693933
s * s
# -8863423146896543127
print(type(s))
# numpy.int64
py_s = int(s)
py_s * py_s
# 9583320926813008489
```

In Python, Integers are implemented as “long”.
In NumPy, Integers are implemented as np.int64.

Digital Representation of Data

```
# using array-scalar type
import numpy as np
dt = np.dtype(np.int32)
print dt
```

| Sr.No. | Data Types & Description |
|--------|--|
| 1 | bool_ Boolean (True or False) stored as a byte |
| 2 | int_ Default integer type (same as C long; normally either int64 or int32) |
| 3 | intc Identical to C int (normally int32 or int64) |
| 4 | intp Integer used for indexing (same as C ssize_t; normally either int32 or int64) |
| 5 | int8 Byte (-128 to 127) |
| 6 | int16 Integer (-32768 to 32767) |

Encoding Integers

Unsigned

$$B2U(X) = \sum_{i=0}^{w-1} x_i \cdot 2^i$$

Two's Complement

$$B2T(X) = -x_{w-1} \cdot 2^{w-1} + \sum_{i=0}^{w-2} x_i \cdot 2^i$$

Sign Bit

```
short int x = 15213;  
short int y = -15213;
```

| | Decimal | Hex | Binary |
|---|---------|-------|-------------------|
| x | 15213 | 3B 6D | 00111011 01101101 |
| y | -15213 | C4 93 | 11000100 10010011 |

Two-complement: Simple Example

$$\begin{array}{rccccc} & -16 & 8 & 4 & 2 & 1 \\ 10 = & 0 & 1 & 0 & 1 & 0 \end{array} \quad 8+2 = 10$$

$$\begin{array}{rccccc} & -16 & 8 & 4 & 2 & 1 \\ -10 = & 1 & 0 & 1 & 1 & 0 \end{array} \quad -16+4+2 = -10$$

Two-complement Encoding Example (Cont.)

| | | | |
|------------|---------|----------|----------|
| x = | 15213: | 00111011 | 01101101 |
| y = | -15213: | 11000100 | 10010011 |

| Weight | 15213 | -15213 | Sum | 15213 | -15213 |
|--------|-------|--------|-----|-------|--------|
| 1 | 1 | 1 | | 1 | 1 |
| 2 | 0 | 0 | | 1 | 2 |
| 4 | 1 | 4 | | 0 | 0 |
| 8 | 1 | 8 | | 0 | 0 |
| 16 | 0 | 0 | | 1 | 16 |
| 32 | 1 | 32 | | 0 | 0 |
| 64 | 1 | 64 | | 0 | 0 |
| 128 | 0 | 0 | | 1 | 128 |
| 256 | 1 | 256 | | 0 | 0 |
| 512 | 1 | 512 | | 0 | 0 |
| 1024 | 0 | 0 | | 1 | 1024 |
| 2048 | 1 | 2048 | | 0 | 0 |
| 4096 | 1 | 4096 | | 0 | 0 |
| 8192 | 1 | 8192 | | 0 | 0 |
| 16384 | 0 | 0 | | 1 | 16384 |
| -32768 | 0 | 0 | | 1 | -32768 |

Numeric Ranges

- Unsigned Values

- $UMin = 0$
• 000...0
- $UMax = 2^w - 1$
• 111...1

- Two's Complement Values

- $TMin = -2^{w-1}$
100...0
- $TMax = 2^{w-1} - 1$
011...1
- Minus 1
111...1

Values for $W = 16$

| | Decimal | Hex | Binary |
|------|---------|-------|-------------------|
| UMax | 65535 | FF FF | 11111111 11111111 |
| TMax | 32767 | 7F FF | 01111111 11111111 |
| TMin | -32768 | 80 00 | 10000000 00000000 |
| -1 | -1 | FF FF | 11111111 11111111 |
| 0 | 0 | 00 00 | 00000000 00000000 |

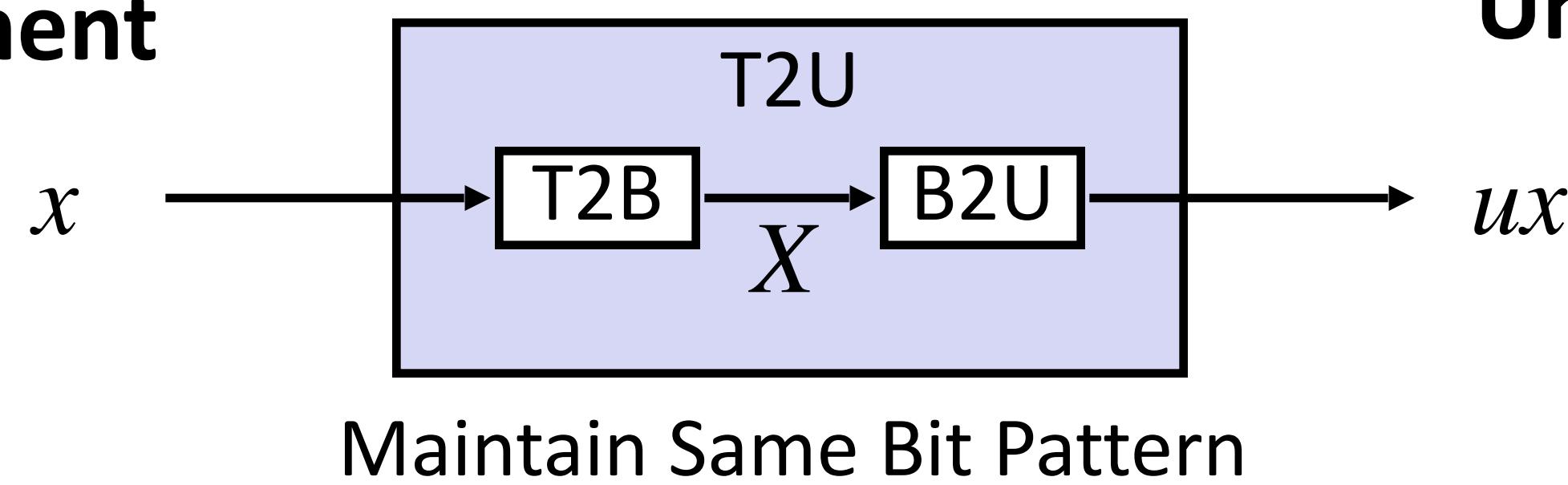
Unsigned & Signed Numeric Values

- **Equivalence**
 - Same encodings for nonnegative values
- **Uniqueness**
 - Every bit pattern represents unique integer value
 - Each representable integer has unique bit encoding
- \Rightarrow **Can Invert Mappings**
 - $U2B(x) = B2U^{-1}(x)$
 - Bit pattern for unsigned integer
 - $T2B(x) = B2T^{-1}(x)$
 - Bit pattern for two's comp integer

| X | $B2U(X)$ | $B2T(X)$ |
|------|----------|----------|
| 0000 | 0 | 0 |
| 0001 | 1 | 1 |
| 0010 | 2 | 2 |
| 0011 | 3 | 3 |
| 0100 | 4 | 4 |
| 0101 | 5 | 5 |
| 0110 | 6 | 6 |
| 0111 | 7 | 7 |
| 1000 | 8 | -8 |
| 1001 | 9 | -7 |
| 1010 | 10 | -6 |
| 1011 | 11 | -5 |
| 1100 | 12 | -4 |
| 1101 | 13 | -3 |
| 1110 | 14 | -2 |
| 1111 | 15 | -1 |

Mapping Between Signed & Unsigned

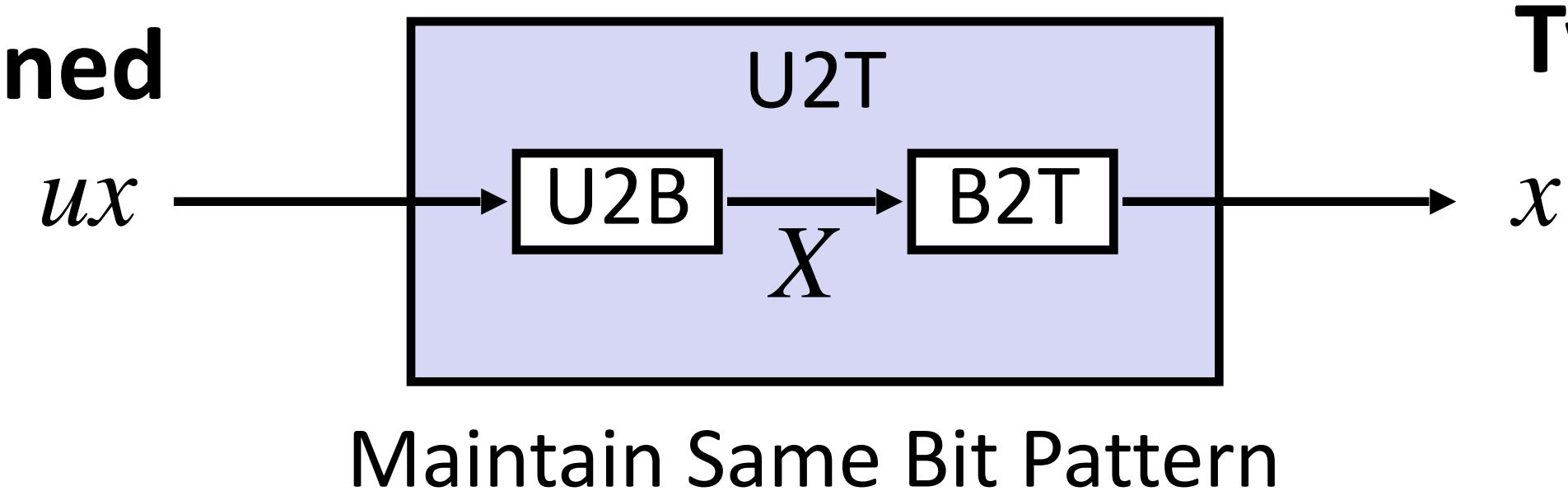
Two's Complement



Unsigned

Maintain Same Bit Pattern

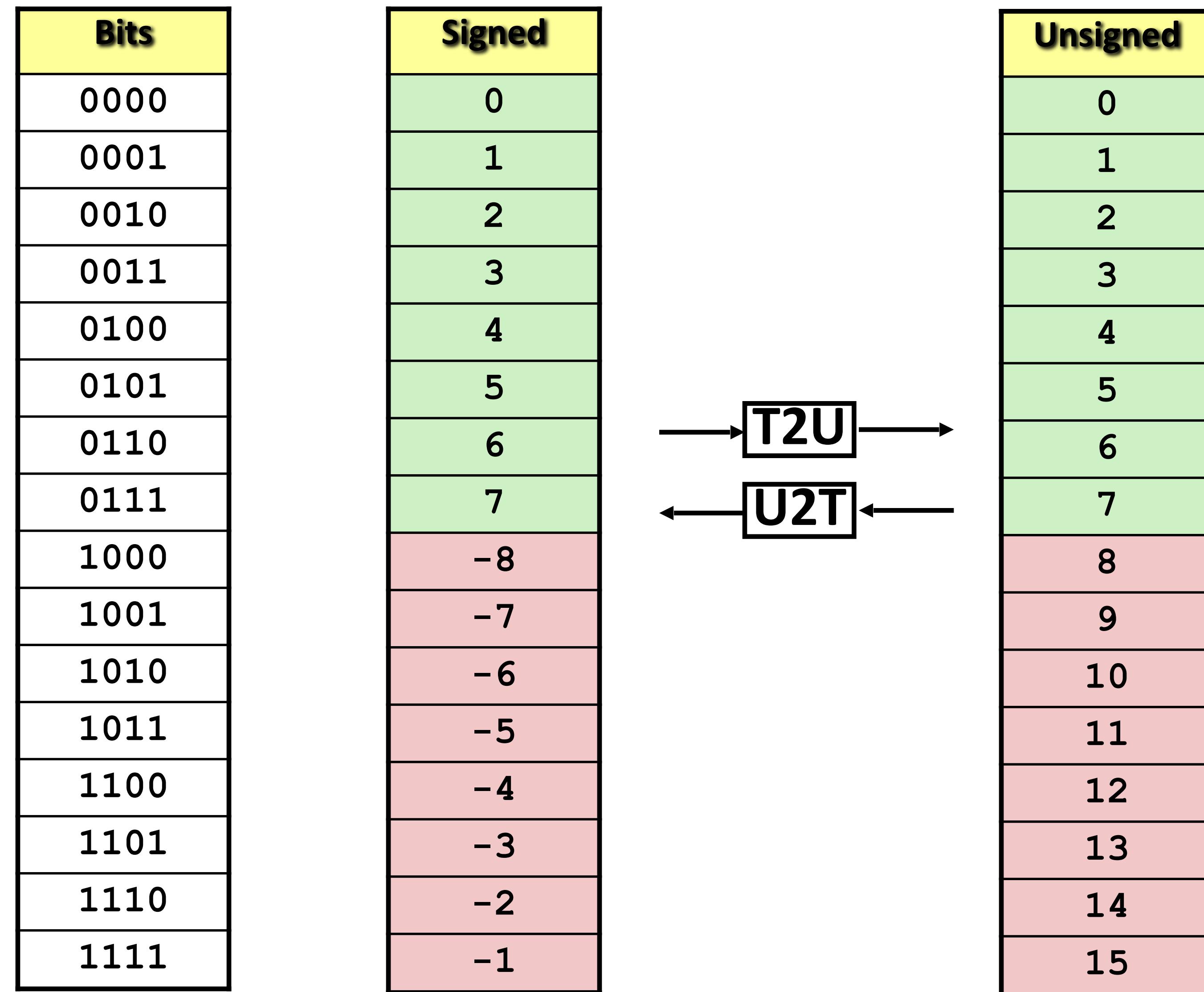
Unsigned



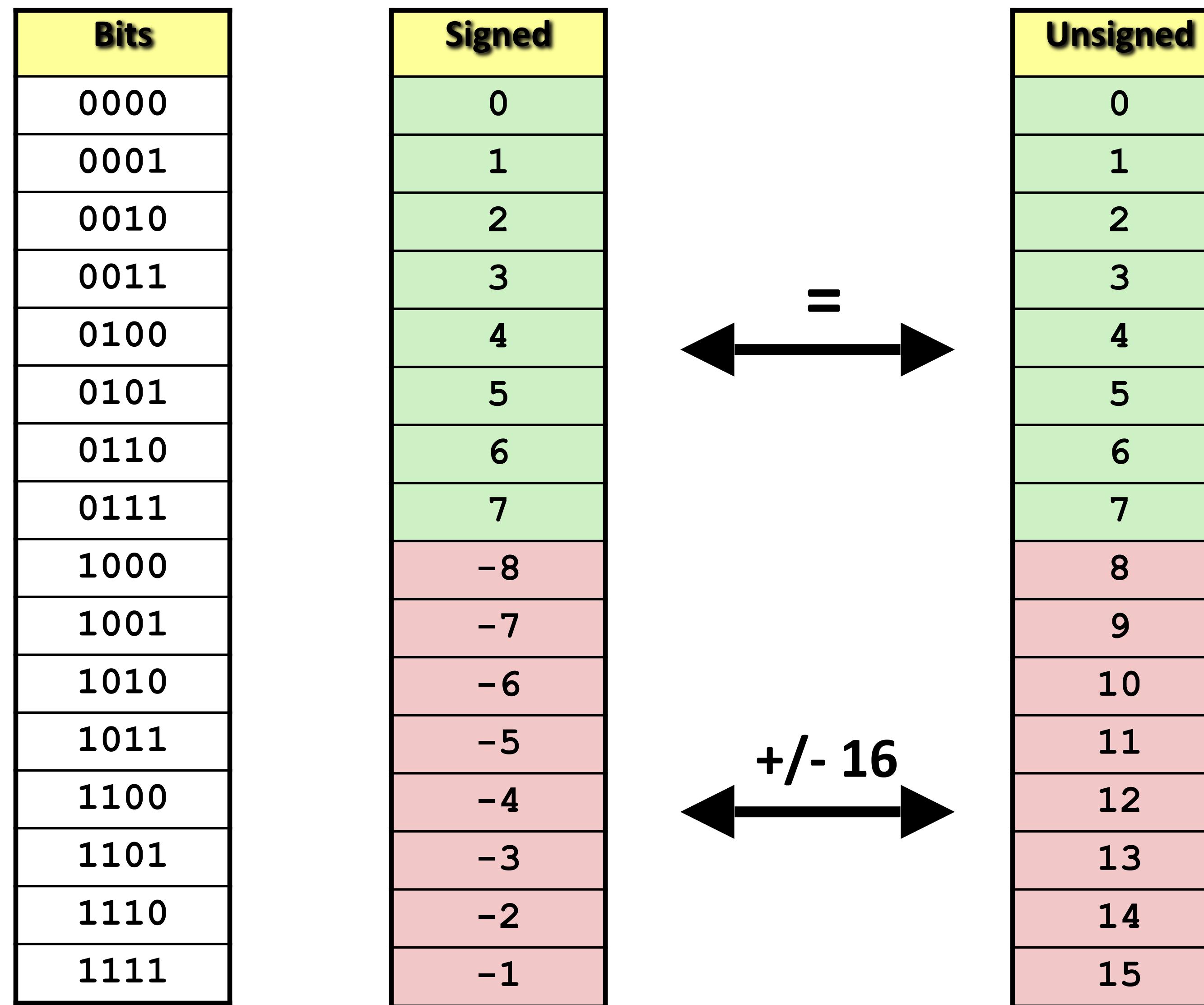
Two's Complement

Maintain Same Bit Pattern

Mapping Signed ↔ Unsigned

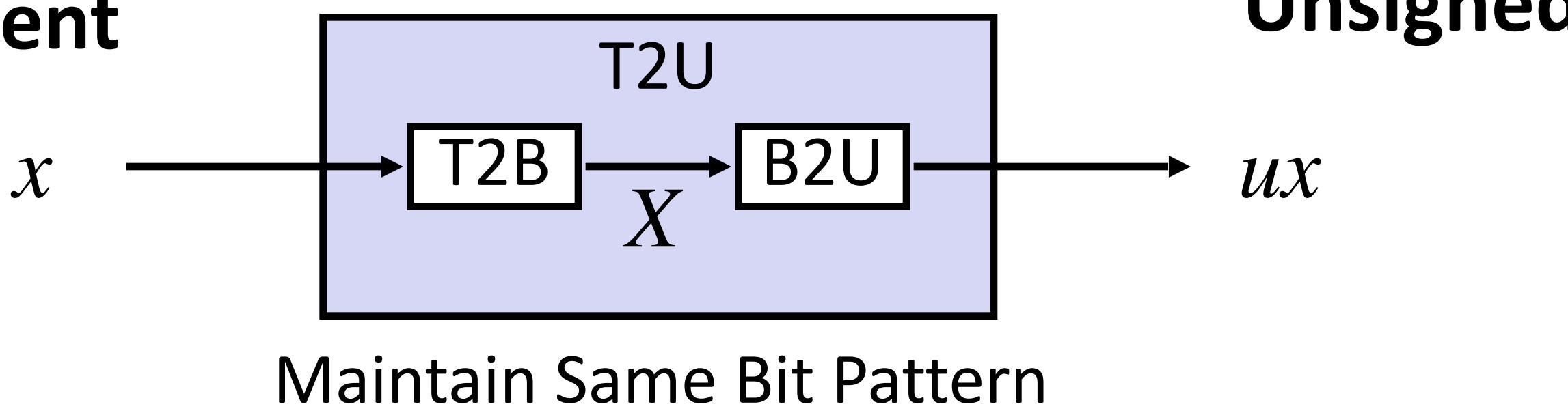


Mapping Signed \leftrightarrow Unsigned

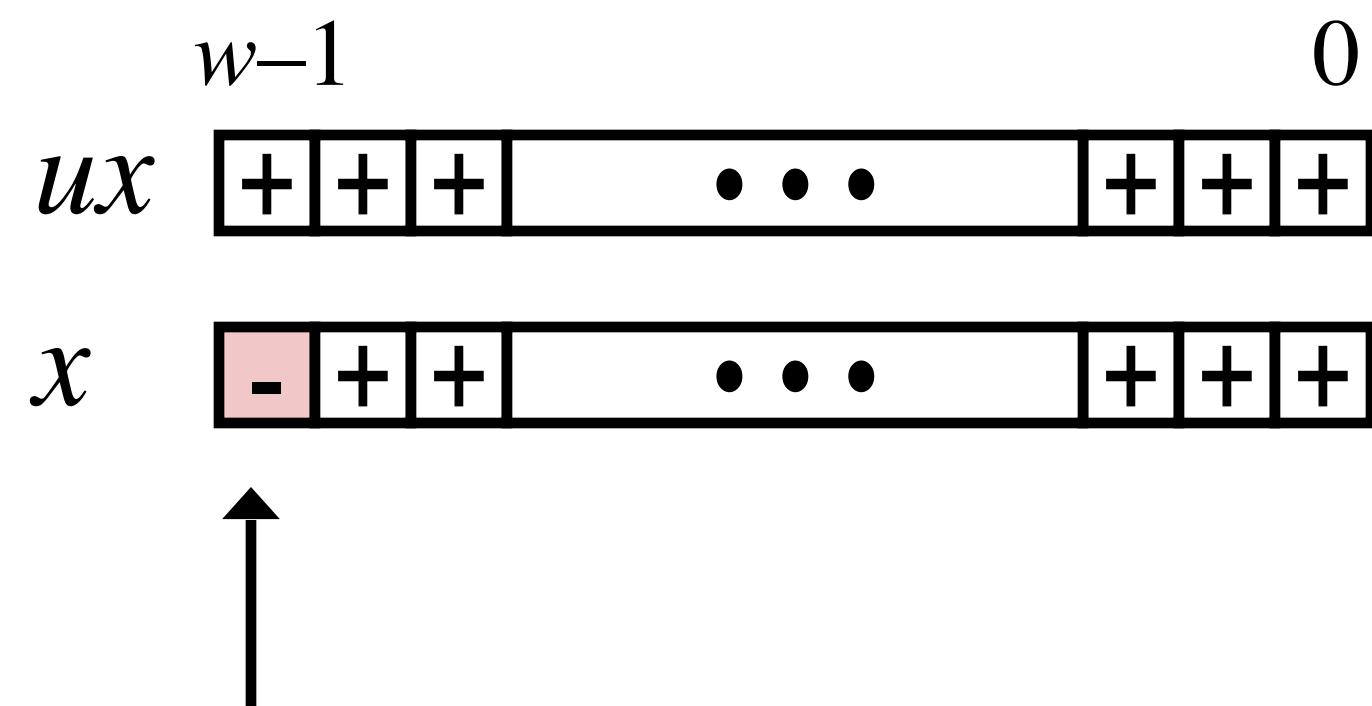


Relation between Signed & Unsigned

Two's Complement



Unsigned



**Large negative weight
becomes
Large positive weight**

Digital Representation of Data

- Representing **Character (char)** and **String**:
 - Letters, numerals, punctuations, etc.
 - A string is typically just a variable-sized array of char
 - C *char* is 1B; Java *char* is 2B; Python does not have a char type (use *str* or *bytes*)
 - American Standard Code for Information Interchange (ASCII) for encoding characters; initially 7-bit; later extended to 8-bit
 - Examples: ‘A’ is 61, ‘a’ is 97, ‘@’ is 64, ‘!’ is 33, etc.
 - *Unicode UTF-8* is now common, subsumes ASCII; 4B for ~1.1 million “code points” incl. many other language scripts, math symbols, emojis, etc. ☺

Other type casting

```
string = "56"
number = 44

# Converting the string into an integer number.
string_number = int(string)

sum_of_numbers = number + string_number
print("The Sum of both the numbers is: ", sum_of_numbers)
```

```
The Sum of both the numbers is: 100
```

Other type casting

```
string = "01101"
floating_number = 45.20

# Converting the string as well as integer number into floating point number.
new_str = int(string)
new_number = int(floating_number)
print("String converted into integer: ", new_str)
print("The floating-point number converted into integer: ", new_number)
```

```
String converted into integer: 1101
The floating-point number converted into integer: 45
```

Data Type conversion examples

| | | | |
|---------------|----------|-----------------|----------|
| int(10) | => 10 | float(10) | => 10.0 |
| int(10.1) | => 10 | float(10.1) | => 10.1 |
| int('10') | => 10 | float('10') | => 10.0 |
| int('10.1') | => Error | float('10.1') | => 10.1 |
| int('kitten') | => Error | float('kitten') | => Error |

| | |
|---------------|-------------|
| str(10) | => '10' |
| str(10.1) | => '10.1' |
| str('kitten') | => 'kitten' |

Digital Representation of Data

- All digital objects are *collections* of basic data types (bytes, integers, floats, and characters)
 - SQL dates/timestamp: string (w/ known format)
 - ML feature vector: *array* of floats (w/ known length)
 - Neural network weights: *set* of multi-dimensional *arrays* (matrices or tensors) of floats (w/ known dimensions)
 - Graph: an *abstract data type* (ADT) with *set* of vertices (say, integers) and *set* of edges (*pair* of integers)
 - Program in PL, SQL query: string (w/ grammar)
 - Other data structures or digital objects?

Review Questions

- Why do computers use binary digits?
- How many integers can you represent with 5 bits?
- How many bits do you need to represent 5 integers?
- What is the hexadecimal for 20_{10} ?
- Why do we need a float standard?
- Why should a data scientist know about float formats?
- What does “lower precision” mean for a float weight in DL?
- Why is serialization needed?