Review

- Python basics
  - Interpreter, data types (int, float, str, bool), memory operation, namespace
  - Functions, modules, classes, and methods
  - Control, grouping, and repetition
  - File I/O
- Object-oriented programming
- Data structures
  - Lists, sets, tuples, and dictionaries
    - Arrays and hashing
  - Linked lists, queues, stacks, binary search trees, general trees, and graphs
- Algorithms
  - Big O, search, sort, and traversal

Bits, Data Types, and Operations

Lecture 20

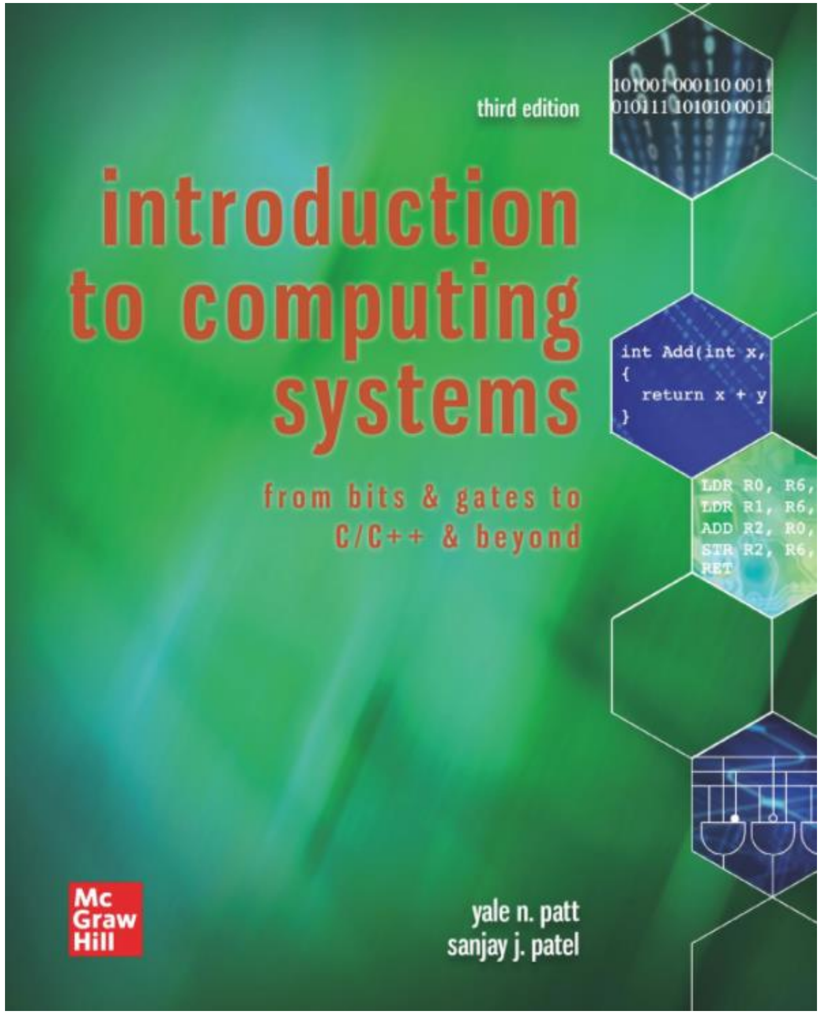
Hyung-Sin Kim

SNU Graduate School of Data Science

Python is a high-level, human-friendly language

Now it is time to explore the low level
to understand computers!

Textbook


third edition
introduction to computing systems
from bits & gates to C/C++ & beyond
yale n. patt
sanjay j. patel
Mc Graw Hill

Contents

Bits

Computing Bootcamp

We have seen many data types (int, float, str, bool).

***Today's main question**:*

How does a computer represent these data types?

Bits (Binary Digits) – What and Why?

- You write a program in a computer language like Python

- When a computer executes this program, it actually means that **electrons** in the computer move in a specific way
  - A computer has billions of very tiny, very fast devices that control the movement of those electrons

- These devices react to the **presence** or **absence** of voltages in electronic circuits (rather than a specific voltage value)
  - Simply detecting presence/absence of voltages is much easier than measuring exact voltage values (120 V, 115 V, 118.6 V …)
  - Making a device react to the actual, exact voltages is very complex and error-prone

Bits (Binary Digits) – What and Why?

- Why is detecting presence/absence of a voltage easier?
  - A circuit does **NOT** have to detect **absolute** presence or absence of a voltage (does not need to be very precise!)

- **Detection with <u>margin</u>:** Rough detection when a computer expects 0 V for absence and 2.9 V for presence
  - A circuit detects presence of a voltage if a voltage is **far from** 0 (e.g., 2.4~2.9 V)
  - A circuit detects absence of a voltage if a voltage is **close** to 0 (e.g., 0~0.5 V)

Bits (Binary Digits) – What and Why?

- Symbolic representation of the two states
  - **1**: Presence of a voltage
  - **0**: Absence of a voltage

- Each 0/1 is called "**bit**," a shortened form of **binary digit**
  - With a bit, now we can represent **two values** successfully!

- But only two values are not enough to do useful tasks!
  - We need to identify a large number of distinct states (values)
  - How?

Computers represent useful information
*by using **multiple bits**!*

***Two bits** can represent four states: 00, 01, 10, 11*
***N bits** can represent 2^N states*

Ok! Then the remaining question is:
*"How does a computer represent **data types***
*using **multiple bits**?"*

Data types

Computing Bootcamp

Data Types

- How can we represent a value “5” by using bits?: There are many ways!

- Unary system (bit counting, **not** using bit-position information)
  - 5 = 11111 (five 1s)
  - We need **too many** bits for representing a large value
- Using bit-position information
  - 5 = 101 ($1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0$)

- Data type
  - A particular **representation** of information in bits
  - There should be **operations** in a computer that can operate on information encoded in that representation

Boolean Types – Values

- A data type that represents two logical values
  - **1**: True
  - **0**: False

- Bit vectors (a sequence of logical variables)
  - Each bit position represents true/false of a specific condition
  - Ex.) A bit vector for GSDS graduation
    - Freshman: 0000
    - Qual Pass!: 0100
    - 4th semester: 0111

Thesis?	Qual?	Units?	Required courses?
0	0	0	0

Boolean Types – Operations

- Operations with **logical variables**

Input A	NOT (~)
0	1
1	0

Input A	Input B	OR (|)
0	0	0
0	1	1
1	0	1
1	1	1

Input A	Input B	AND (&)
0	0	0
0	1	0
1	0	0
1	1	1

Input A	Input B	XOR (^)
0	0	0
0	1	1
1	0	1
1	1	0

Boolean Types – Operations

- Operations with **bit vectors**

Input A	NOT (~)
0110	1001
1111	0000

Input A	Input B	OR (|)
0111	1110	1111
0000	1010	1010
0101	0011	0111
1110	1111	1111

Input A	Input B	AND (&)
0111	1110	0110
0000	1010	0000
0101	0011	0001
1110	1111	1110

Input A	Input B	XOR (^)
0111	1110	1001
0000	1010	1010
0101	0011	0110
1110	1111	0001

Integer representation

- Unsigned integers
- Signed integers
- Operations

Representing Integers
(maybe more complex than you thought)

Unsigned Integers

- A data type that represents <u>positive integers and zero</u> (no negative integers)
  - N bits can represent unsigned integers from 0 to $2^N - 1$

Value	2-nd Bit	1-st Bit	0-th Bit
0	0	0	0
1	0	0	1
2	0	1	0
3	0	1	1
4	1	0	0
5	1	0	1
6	1	1	0
7	1	1	1

- Operations: Addition, subtraction, multiplication, division, etc… with **Base 2**

```
                    carry
  10010      10010        1111
+  1001    +  1011     +     1
  11011      11101       10000

             10111
           +   111
```

Signed Integers – Signed Magnitude

- A data type that represents positive/negative integers and zero
  - N bits can represent signed integers from $-2^{N-1}+1$ to $2^{N-1}-1$

- Most significant bit (the leftmost bit) to represent **sign**
  - 0: Positive
  - 1: Negative

- Approach 1: **Signed magnitude**
  - Use other bits just as unsigned integers
  - 2 = 0 10
  - -2 = 1 10

Value	2-nd Bit	1-st Bit	0-th Bit
0	0	0	0
1	0	0	1
2	0	1	0
3	0	1	1
-0	1	0	0
-1	1	0	1
-2	1	1	0
-3	1	1	1

Signed Integers – 1's Complement

- A data type that represents positive/negative integers and zero
  - N bits can represent signed integers from $-2^{N-1}+1$ to $2^{N-1}-1$

- Most significant bit (the leftmost bit) to represent **sign**
  - 0: Positive
  - 1: Negative

- Approach 2: **1's complement**
  - Flip the other bits
  - 2 = 0 10
  - -2 = 1 01

Value	2-nd Bit	1-st Bit	0-th Bit
0	0	0	0
1	0	0	1
2	0	1	0
3	0	1	1
-3	1	0	0
-2	1	0	1
-1	1	1	0
-0	1	1	1

Signed Integers – 2's complement

- Problems with signed magnitude and 1's complement
  - Two representations of zero (+0 and -0)
  - Complex rules for operations: complex to design arithmetic circuits

- We want to represent negative integers so that logic circuits become as simple as possible!

- **2's complement** represent a negative value (-X) so that X + (-X) = 0 with "normal" binary addition (ignoring carry out)

```
  00101  (5)          01001  (9)
+______  (-5)       +______  (-9)
  00000  (0)          00000  (0)
```

Signed Integers – 2's complement

- Problems with signed magnitude and 1's complement
  - Two representations of zero (+0 and -0)
  - Complex rules for operations: complex to design arithmetic circuits

- We want to represent negative integers so that logic circuits become as simple as possible!

- **2's complement** represent a negative value (-X) so that X + (-X) = 0 with "normal" binary addition (ignoring carry out)

$$\begin{array}{rl} 00101 & (5) \\ +\ \boxed{11011} & (-5) \\ \hline 00000 & (0) \end{array} \qquad \begin{array}{rl} 01001 & (9) \\ +\ \boxed{10111} & (-9) \\ \hline 00000 & (0) \end{array}$$

Signed Integers – 2's complement

- Representing a negative integer by using 2's complement
  - Represent its absolute value as an unsigned integer
  - Flip every bit (i.e., take the 1's complement)
  - Add one

-5 → 00101 → 11010 → **11011**

Value	2-nd Bit	1-st Bit	0-th Bit
0	0	0	0
1	0	0	1
2	0	1	0
3	0	1	1
-4	1	0	0
-3	1	0	1
-2	1	1	0
-1	1	1	1

- The most significant bit has weight -2^{N-1}

Arithmetic Operation – Addition

- 2's complement addition is just binary addition
  - Assumption
    - All integers have the same number of bits (e.g., 8 bits)
    - The result can still be represented as the same number of bits (e.g., 8 bits)
  - Ignore carry out (e.g., 9-th bit)

```
    01101000 (104)
+   11110000 (-16)
--------------------
    01011000 ( 88)
```

```
    11110110 (-10)
+   11110111 (-9)
--------------------
    11101101 (-19)
```

Arithmetic Operation – Subtraction

- 2's complement subtraction
  - Assumption
    - All integers have the same number of bits (e.g., 8 bits)
    - The result can still be represented as the same number of bits (e.g., 8 bits)
  - Step 1) **Negate** the subtrahend (2nd number)
  - Step 2) Perform Addition

```
    01101000 (104)          11110110 (-10)
-   00010000 (16)       -   11110111 (-9)
-----------------       -----------------
    01101000 (104)          11110110 (-10)
+   11110000 (-16)      +   00001001 ( 9)
-----------------       -----------------
    01011000 ( 88)          11111111 (-1)
```

Arithmetic Operation – Sign Extension

- 2's complement addition when two integers have **different lengths**
  - Extend the smaller number's bit length
    - Adding 0s for positive integers

```
  01101000 (104)
+     0100 (4)
---------------
  01101000 (104)
+ 00000100 (4)
---------------
  01101100 (108)
```

```
  01101000 (104)
+     1100 (-4)
---------------
  01101000 (104)
+ 00001100 (8??)
---------------
```

Zero padding does **NOT** work for negative integers!

Arithmetic Operation – Sign Extension

- 2's complement addition when two integers have **different lengths**
  - Extend the smaller number's bit length
    - Adding 0s for positive integers
    - Adding 1s for negative integers

```
   01101000 (104)          01101000 (104)
+      0100 (4)         +      1100 (-4)
--------------          --------------
   01101000 (104)          01101000 (104)
+  00000100 (4)         +  11111100 (-4)
--------------          --------------
   01101100 (108)          01100100 (100)
```

Arithmetic Operation – Overflow

- Overflow: If operands are too big, the result sometime cannot be represented as an n-bit 2’s complement number

- It is important to estimate and detect overflow
  - It can happen when signs of both operands are the same
  - It can be detected by testing the most significant bit (the sign bit)

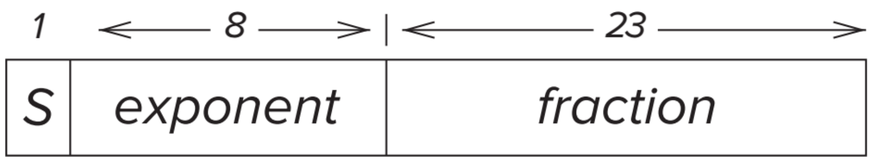
	01101000 (104)		10010110 (-104)
+	00100001 (33)	+	11011111 (-33)
	10001001 (-119)		01110101 (117)

Other representations

Floating Point

- Representation (binary number)
  - $N = (-1)^{\boxed{S}} \times 1.\boxed{fraction} \times 2^{\boxed{exponent}-127},\ 1 \le exponent \le 254$

- Embed the three information in a 32-bit data type
  - 1 bit for the sign (positive or negative)
  - 8 bits for the range (the exponent field)
  - 23 bits for precision (the fraction field)

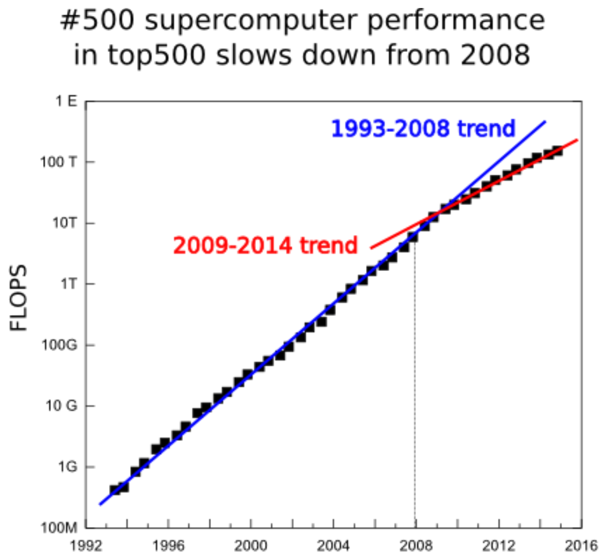


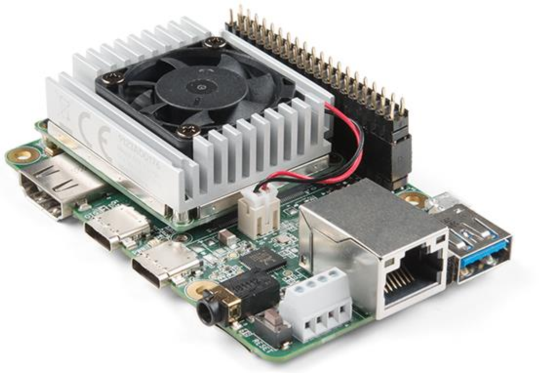
- Example: $\boxed{0}\boxed{01111011}\boxed{00000000000000000000000}$
  - S = 0, exponent = 01111011 (123), fraction = 00000000000000000000000
  - $1.00000000000000000000000 \times 2^{123-127} = \frac{1}{16}$

Floating Point

- Floating point arithmetic is more complex than integer (fixed-point) arithmetic
  - FLOPS (floating point operations per second) is a representative measure of computer performance

- Trade-off
  - **Development cost**: It is easier to design an algorithm with floating point numbers
    - Need to handle quantization noise to design an algorithm with fixed-point numbers
  - **Execution cost**: It is faster and more energy efficient to execute an algorithm with fixed-point numbers





ASCII

- 8-bit code to represent **characters**
  - Used for communication between the main computer processing unit and input/output devices (e.g., keyboards and monitors)
- Greatly simplifies the interface between
  - a keyboard made by one company
  - a computer made by another company
  - a monitor made by a third company

Dec	Bin	Hex	Char	Dec	Bin	Hex	Char	Dec	Bin	Hex	Char	Dec	Bin	Hex	Char
0	0000 0000	00	[NUL]	32	0010 0000	20	space	64	0100 0000	40	@	96	0110 0000	60	`
1	0000 0001	01	[SOH]	33	0010 0001	21	!	65	0100 0001	41	A	97	0110 0001	61	a
2	0000 0010	02	[STX]	34	0010 0010	22	"	66	0100 0010	42	B	98	0110 0010	62	b
3	0000 0011	03	[ETX]	35	0010 0011	23	#	67	0100 0011	43	C	99	0110 0011	63	c
4	0000 0100	04	[EOT]	36	0010 0100	24	$	68	0100 0100	44	D	100	0110 0100	64	d
5	0000 0101	05	[ENQ]	37	0010 0101	25	%	69	0100 0101	45	E	101	0110 0101	65	e
6	0000 0110	06	[ACK]	38	0010 0110	26	&	70	0100 0110	46	F	102	0110 0110	66	f
7	0000 0111	07	[BEL]	39	0010 0111	27	'	71	0100 0111	47	G	103	0110 0111	67	g
8	0000 1000	08	[BS]	40	0010 1000	28	(	72	0100 1000	48	H	104	0110 1000	68	h
9	0000 1001	09	[TAB]	41	0010 1001	29	)	73	0100 1001	49	I	105	0110 1001	69	i
10	0000 1010	0A	[LF]	42	0010 1010	2A	*	74	0100 1010	4A	J	106	0110 1010	6A	j
11	0000 1011	0B	[VT]	43	0010 1011	2B	+	75	0100 1011	4B	K	107	0110 1011	6B	k
12	0000 1100	0C	[FF]	44	0010 1100	2C	,	76	0100 1100	4C	L	108	0110 1100	6C	l
13	0000 1101	0D	[CR]	45	0010 1101	2D	-	77	0100 1101	4D	M	109	0110 1101	6D	m
14	0000 1110	0E	[SO]	46	0010 1110	2E	.	78	0100 1110	4E	N	110	0110 1110	6E	n
15	0000 1111	0F	[SI]	47	0010 1111	2F	/	79	0100 1111	4F	O	111	0110 1111	6F	o
16	0001 0000	10	[DLE]	48	0011 0000	30	0	80	0101 0000	50	P	112	0111 0000	70	p
17	0001 0001	11	[DC1]	49	0011 0001	31	1	81	0101 0001	51	Q	113	0111 0001	71	q
18	0001 0010	12	[DC2]	50	0011 0010	32	2	82	0101 0010	52	R	114	0111 0010	72	r
19	0001 0011	13	[DC3]	51	0011 0011	33	3	83	0101 0011	53	S	115	0111 0011	73	s
20	0001 0100	14	[DC4]	52	0011 0100	34	4	84	0101 0100	54	T	116	0111 0100	74	t
21	0001 0101	15	[NAK]	53	0011 0101	35	5	85	0101 0101	55	U	117	0111 0101	75	u
22	0001 0110	16	[SYN]	54	0011 0110	36	6	86	0101 0110	56	V	118	0111 0110	76	v
23	0001 0111	17	[ETB]	55	0011 0111	37	7	87	0101 0111	57	W	119	0111 0111	77	w
24	0001 1000	18	[CAN]	56	0011 1000	38	8	88	0101 1000	58	X	120	0111 1000	78	x
25	0001 1001	19	[EM]	57	0011 1001	39	9	89	0101 1001	59	Y	121	0111 1001	79	y
26	0001 1010	1A	[SUB]	58	0011 1010	3A	:	90	0101 1010	5A	Z	122	0111 1010	7A	z
27	0001 1011	1B	[ESC]	59	0011 1011	3B	;	91	0101 1011	5B	[	123	0111 1011	7B	{
28	0001 1100	1C	[FS]	60	0011 1100	3C	<	92	0101 1100	5C	\	124	0111 1100	7C	|
29	0001 1101	1D	[GS]	61	0011 1101	3D	=	93	0101 1101	5D	]	125	0111 1101	7D	}
30	0001 1110	1E	[RS]	62	0011 1110	3E	>	94	0101 1110	5E	^	126	0111 1110	7E	~
31	0001 1111	1F	[US]	63	0011 1111	3F	?	95	0101 1111	5F	_	127	0111 1111	7F	[DEL]

Hexagonal

- A method to represent binary strings (0s and 1s…) for **human convenience** when there are too many bits …
  - Long binary strings can cause many copying errors!: 0011110101101110
- Group four bits (0~15) and represent the number as a hex digit
  - 0 1 2 3 4 5 6 7 8 9 A B C D E F
- Example

0011 1101 0110 1110

0011 1101 0110 1110

3 D 6 E

3D6E

Summary

Summary

- Exact value vs. Presence/Absence
- Bit, multiple bits, and data types
- Logical variables, bit vectors, and operations
- Unsigned integers, signed integers, and operations
- Floating point, ASCII, and Hexagonal

Q&A

Any questions?

Thanks!