



COMP110: Principles of Computing

## **3: Data Types**

**Binary notation**



there are 10 types  
of people in the  
world: people who  
understand binary,  
and people who  
have friends

# How we write numbers

- ▶ We write numbers in **base 10**
- ▶ We have 10 **digits**: 0, 1, 2, ..., 8, 9
- ▶ When we write 6397, we mean:
  - ▶ Six thousand, three hundred and ninety seven
  - ▶ (Six thousands) and (three hundreds) and (nine tens) and (seven)
  - ▶  $(6 \times 1000) + (3 \times 100) + (9 \times 10) + (7)$
  - ▶  $(6 \times 10^3) + (3 \times 10^2) + (9 \times 10^1) + (7 \times 10^0)$
  - ▶

Thousands	Hundreds	Tens	Units
6	3	9	7

# Binary

- ▶ Binary notation works the same, but is **base 2** instead of **base 10**
- ▶ We have 2 **digits**: 0, 1
- ▶ When we write 10001011 in binary, we mean:
$$\begin{aligned}& (1 \times 2^7) + (0 \times 2^6) + (0 \times 2^5) + (0 \times 2^4) \\& + (1 \times 2^3) + (0 \times 2^2) + (1 \times 2^1) + (1 \times 2^0) \\& = 2^7 + 2^3 + 2^1 + 2^0 \\& = 128 + 8 + 2 + 1 \text{ (base 10)} \\& = 139 \text{ (base 10)}\end{aligned}$$

# Why binary?

- ▶ Modern computers are **digital**
- ▶ Based on the flow of current in a circuit being either **on** or **off**
- ▶ Hence it is natural to store and operate on numbers in base 2
- ▶ The binary digits 0 and 1 correspond to **off** and **on** respectively

# Converting to binary

[https://www.youtube.com/watch?v=OezK\\_zTyvAQ](https://www.youtube.com/watch?v=OezK_zTyvAQ)

# Bits, bytes and words

- ▶ A **bit** is a binary digit
  - ▶ Can store a 0 or 1 (i.e. a boolean value)
  - ▶ The smallest possible unit of information
- ▶ A **byte** is 8 **bits**
  - ▶ Can store a number between 0 and 255 in binary
- ▶ A **word** is the number of bits that the CPU works with at once
  - ▶ 32-bit CPU: 32 bits = 1 word
  - ▶ 64-bit CPU: 64 bits = 1 word
- ▶ An  $n$ -bit word can store a number between 0 and  $2^n - 1$ 
  - ▶  $2^{16} - 1 = 65,535$
  - ▶  $2^{32} - 1 = 4,294,967,295$
  - ▶  $2^{64} - 1 = 18,446,744,073,709,551,615$



# Other units

- ▶ A **nibble** is 4 **bits**
- ▶ A **kilobyte** is 1000 or 1024 **bytes**
  - ▶  $10^3 = 1000 \approx 1024 = 2^{10}$
- ▶ A **megabyte** is 1000 or 1024 **kilobytes**
- ▶ A **gigabyte** is 1000 or 1024 **megabytes**
- ▶ A **terabyte** is 1000 or 1024 **gigabytes**
- ▶ ...

# Addition with carry

In base 10:

$$\begin{array}{rcccc} & 1 & 2 & 3 & 4 \\ + & 5 & 6_1 & 7_1 & 8 \\ \hline & 6 & 9 & 1 & 2 \end{array}$$

# Addition with carry

In base 2:

$1 + 1 = 10$	$1 + 1 + 1 = 11$
--------------	------------------

	0	1	1	0	1	1	1	0
+	0 <sub>1</sub>	0 <sub>1</sub>	1	0 <sub>1</sub>	0 <sub>1</sub>	1 <sub>1</sub>	1	1
<hr/>								
	1	0	0	1	0	1	0	1

# Hexadecimal notation

- ▶ Other number bases than 2 and 10 are also useful

- ▶ Hexadecimal is **base 16**

- ▶ Uses extra digits:

- ▶ A=10, B=11, ..., F=15

Hex	Dec	Hex	Dec	Hex	Dec
00	0	10	16	F0	240
01	1	11	17	F1	241
⋮	⋮	⋮	⋮	⋮	⋮
09	9	19	25	F9	249
0A	10	1A	26	FA	250
0B	11	1B	27	FB	251
0C	12	1C	28	FC	252
0D	13	1D	29	FD	253
0E	14	1E	30	FE	254
0F	15	1F	31	FF	255

# **Numeric types**

# Integers

- ▶ An **integer** is a whole number — positive, negative or zero
- ▶ Python type: `int`
- ▶ In most languages, `int` is limited to 32 or 64 bits
- ▶ Python uses **big integers** — number of bits expands automatically to fit the value to be stored
- ▶ Stored in memory using binary notation, with 2's complement for negative values

# Floating point numbers

- ▶ What about storing non-integer numbers?
- ▶ Usually we use **floating point** numbers
- ▶ Python type: `float`
- ▶ Details on in-memory representation later in the module
- ▶ (Note: `float` in Python 3 has the same precision as `double` in C++/C#/etc)

# Integers vs floating point numbers

- ▶ `int` and `float` are different types!
- ▶ `42` and `42.0` are technically different values
  - ▶ One is an `int`, the other is a `float`
  - ▶ They are stored differently in memory (completely different sequences of bytes)
  - ▶ However `==` etc still know how to compare them sensibly



**String types**

# Strings

- ▶ A **string** represents a sequence of textual characters
- ▶ E.g. "Hello world!"
- ▶ Python type: **str**

# String representation

- ▶ Stored as sequences of **characters** encoded as **integers**
- ▶ Often **null-terminated**
  - ▶ Character number 0 signifies the end of the string

# What is a character?

- ▶ Broadly speaking, a single **printable symbol**
- ▶ There are also some special **non-printable characters**  
e.g. line break

# ASCII

- ▶ American Standard Code for Information Interchange
- ▶ Defines a standard set of 128 characters (7 bits per character)
- ▶ Originally developed in the 1960s for teletype machines, but survives in computing to this day
- ▶ 95 printable characters: upper and lower case English alphabet, digits, punctuation
- ▶ 33 non-printable characters

Hex	Value	Hex	Value	Hex	Value	Hex	Value	Hex	Value	Hex	Value	Hex	Value	Hex	Value
00	NUL	10	DLE	20	SP	30	0	40	@	50	P	60	`	70	p
01	SOH	11	DC1	21	!	31	1	41	A	51	Q	61	a	71	q
02	STX	12	DC2	22	"	32	2	42	B	52	R	62	b	72	r
03	ETX	13	DC3	23	#	33	3	43	C	53	S	63	c	73	s
04	EOT	14	DC4	24	\$	34	4	44	D	54	T	64	d	74	t
05	ENQ	15	NAK	25	%	35	5	45	E	55	U	65	e	75	u
06	ACK	16	SYN	26	&	36	6	46	F	56	V	66	f	76	v
07	BEL	17	ETB	27	'	37	7	47	G	57	W	67	g	77	w
08	BS	18	CAN	28	(	38	8	48	H	58	X	68	h	78	x
09	HT	19	EM	29	)	39	9	49	I	59	Y	69	i	79	y
0A	LF	1A	SUB	2A	*	3A	:	4A	J	5A	Z	6A	j	7A	z
0B	VT	1B	ESC	2B	+	3B	;	4B	K	5B	[	6B	k	7B	{
0C	FF	1C	FS	2C	,	3C	<	4C	L	5C	\	6C	l	7C	
0D	CR	1D	GS	2D	-	3D	=	4D	M	5D	]	6D	m	7D	}
0E	SO	1E	RS	2E	.	3E	>	4E	N	5E	^	6E	n	7E	~
0F	SI	1F	US	2F	/	3F	?	4F	O	5F	_	6F	o	7F	DEL

# ASCII

- ▶ ASCII works OK for English
- ▶ Standards exist to add another 128 characters (taking us to 8 bits per character)
- ▶ E.g. accented characters for European languages, other Western alphabets e.g. Greek, Cyrillic, mathematical symbols
- ▶ However 256 characters isn't enough...

# Unicode

- ▶ Standard character set developed from 1987 to present day
- ▶ Currently defines 143859 characters (Unicode 13.0)
- ▶ First 128 characters are the same as ASCII
- ▶ Covers most of the world's writing systems
- ▶ Also covers mathematical symbols and emoji



# Encoding Unicode

- ▶ **UTF-32** encodes characters as 32-bit integers
- ▶ **UTF-8** encodes characters as 8, 16, 24 or 32-bit integers
  - ▶ 8-bit characters correspond to the first 128 ASCII characters  $\implies$  backwards compatible
  - ▶ More common Unicode characters are smaller  $\implies$  more efficient than UTF-32

# Escape sequences

- ▶ Backslash `\` has a special meaning in string literals — it denotes the start of an **escape sequence**
- ▶ Typically used to write **non-printable characters**
- ▶ Most useful: `"\n"` is a new line
- ▶ How to type a backslash character? Use `"\\"`

# Text files

- ▶ Stored on disk as essentially one long string
- ▶ Line endings are denoted by non-printable characters
  - ▶ Unix format: line feed character (ASCII/UTF-8 character 10, "`\n`")
  - ▶ Windows format: carriage return character (ASCII/UTF-8 character 13) followed by line feed, "`\r\n`"
  - ▶ Most text editors can handle and convert both formats
  - ▶ Most languages allow files to be opened in "text mode" which automatically converts

**Other types**

# Booleans

- ▶ A **boolean** can have one of two values: **true** or **false**
- ▶ Python type: `bool`
- ▶ In Python, we have the keywords `True` and `False`
- ▶ Could be represented by a single bit in memory...
- ▶ ... but since memory is addressed in bytes (or words of multiple bytes), usually represented as an `int` with 0 meaning `False` and any non-zero (e.g. 1) meaning `True`

# Boolean values

- ▶ The `if` statement takes a boolean value as its condition:

```
if x > 10:  
    print(x)
```

- ▶ Variables can also store boolean values:

```
result = (x > 10)    # result now stores True or False  
if result:  
    print(x)
```

# The “None” value

- ▶ Python has a special value `None` which can be used to denote the “absence” of any other value
- ▶ Python type: `NoneType`

# Checking types in Python

- ▶ Call `type()` to check the type of a variable or value
- ▶ Note that `type()` returns a value of type `type`
- ▶ You can use these `type` values like any other value, e.g.

```
if type(x) == int:
    print("x has type int")
elif type(x) == type(y):
    print("x and y have the same type")
```



# Other types

- ▶ **Container** types for collecting several values
  - ▶ `list, tuple, dict, set, ...`
- ▶ **Objects** — a way to define your own types
- ▶ Almost everything in Python is a value with a type
  - ▶ Functions, modules, classes, exceptions, ...

# Boolean Logic

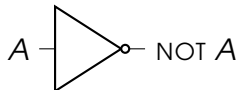
# Boolean logic

- ▶ Works with two values: TRUE and FALSE
- ▶ Foundation of the **digital computer**: represented in circuits as **on** and **off**
- ▶ Representing as 1 and 0 leads to **binary notation**
- ▶ One boolean value = one **bit** of information
- ▶ Programmers use boolean logic for conditions in **if** and **while** statements

# Not

NOT  $A$  is TRUE  
if and only if  
 $A$  is FALSE

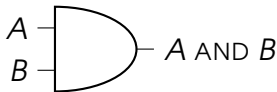
$A$	NOT $A$
FALSE	TRUE
TRUE	FALSE



# And

$A$  AND  $B$  is TRUE  
if and only if  
**both**  $A$  **and**  $B$  are TRUE

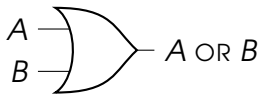
$A$	$B$	$A$ AND $B$
FALSE	FALSE	FALSE
FALSE	TRUE	FALSE
TRUE	FALSE	FALSE
TRUE	TRUE	TRUE



# Or

$A$  OR  $B$  is TRUE  
if and only if  
**either**  $A$  **or**  $B$ , **or both**, are TRUE

$A$	$B$	$A$ OR $B$
FALSE	FALSE	FALSE
FALSE	TRUE	TRUE
TRUE	FALSE	TRUE
TRUE	TRUE	TRUE



# Socratic FALCOMPED

What is the value of

$A \text{ AND } (B \text{ OR } C)$

when

$A = \text{TRUE}$

$B = \text{FALSE}$

$C = \text{TRUE}$

?

# Socratic FALCOMPED

What is the value of

$(\text{NOT } A) \text{ AND } (B \text{ OR } C)$

when

$A = \text{TRUE}$

$B = \text{FALSE}$

$C = \text{TRUE}$

?



# Socratic FALCOMPED

For what values of  $A, B, C, D$  is

$$A \text{ AND NOT } B \text{ AND NOT } (C \text{ OR } D) = \text{TRUE}$$

?

# Socratic FALCOMPED

What is the value of

$A \text{ OR NOT } A$

?

# Socratic FALCOMPED

What is the value of

$A \text{ AND NOT } A$

?

# Socratic FALCOMPED

What is the value of

$A \text{ OR } A$

?

# Socratic FALCOMPED

What is the value of

$A \text{ AND } A$

?

# Writing logical operations

Operation	Python	C family	Mathematics
NOT $A$	<code>not a</code>	<code>!a</code>	$\neg A$ or $\overline{A}$
$A$ AND $B$	<code>a and b</code>	<code>a &amp;&amp; b</code>	$A \wedge B$
$A$ OR $B$	<code>a or b</code>	<code>a    b</code>	$A \vee B$

Other operators can be expressed by combining these

# De Morgan's Laws

$$\text{NOT } (A \text{ OR } B) = (\text{NOT } A) \text{ AND } (\text{NOT } B)$$

$$\text{NOT } (A \text{ AND } B) = (\text{NOT } A) \text{ OR } (\text{NOT } B)$$

Proof: see this week's worksheet

# Truth tables



# Enumeration

- ▶ Since booleans have only two possible values, we can often **enumerate** all possible values of a set of boolean variables
- ▶ For  $n$  variables there are  $2^n$  possible combinations
- ▶ Essentially, all the  $n$ -bit binary numbers
- ▶ A **truth table** enumerates all the possible values of a boolean expression
- ▶ Can be used to prove that two expressions are equivalent

# Truth table example

$(A \text{ OR NOT } B) \text{ AND } C$

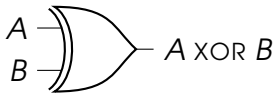
$A$	$B$	$C$	NOT $B$	$A \text{ OR NOT } B$	$(A \text{ OR NOT } B) \text{ AND } C$
FALSE	FALSE	FALSE	TRUE	TRUE	FALSE
FALSE	FALSE	TRUE	TRUE	TRUE	TRUE
FALSE	TRUE	FALSE	FALSE	FALSE	FALSE
FALSE	TRUE	TRUE	FALSE	FALSE	FALSE
TRUE	FALSE	FALSE	TRUE	TRUE	FALSE
TRUE	FALSE	TRUE	TRUE	TRUE	TRUE
TRUE	TRUE	FALSE	FALSE	TRUE	FALSE
TRUE	TRUE	TRUE	FALSE	TRUE	TRUE

**Other logic gates**

# Exclusive Or

$A \text{ XOR } B$  is TRUE  
if and only if  
**either**  $A$  **or**  $B$ , **but not both**, are TRUE

$A$	$B$	$A \text{ XOR } B$
FALSE	FALSE	FALSE
FALSE	TRUE	TRUE
TRUE	FALSE	TRUE
TRUE	TRUE	FALSE



# Socratic FALCOMPED

How can  $A \text{ XOR } B$  be written using the operations  
AND , OR , NOT ?

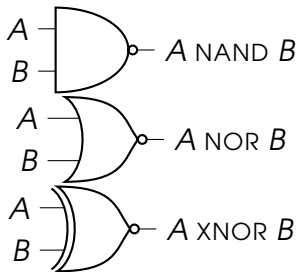
# Negative gates

NAND , NOR , XNOR  
are the **negations** of  
AND , OR , XOR

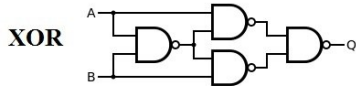
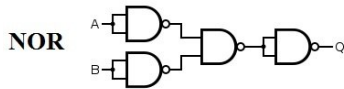
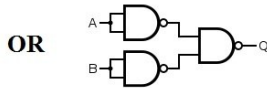
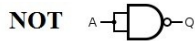
$$A \text{ NAND } B = \text{NOT } (A \text{ AND } B)$$

$$A \text{ NOR } B = \text{NOT } (A \text{ OR } B)$$

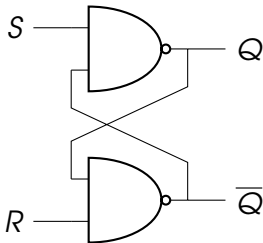
$$A \text{ XNOR } B = \text{NOT } (A \text{ XOR } B)$$



# Any logic gate can be constructed from NAND gates



# What does this circuit do?



- ▶ This is called a **NAND latch**
- ▶ It “remembers” a single boolean value
- ▶ Put a few billion of these together (along with some control circuitry) and you’ve got **memory**!



# NAND gates

- ▶ All arithmetic and logic operations, as well as memory, can be built from NAND gates
- ▶ So an entire computer can be built just from NAND gates!
- ▶ NAND gate circuits are **Turing complete**
- ▶ The same is true of NOR gates

# Workshop activity

- ▶ Split into your **breakout groups** (see Week 2 on LearningSpace, and find the breakout channels in the COMP110 team on Teams)
- ▶ Play <http://nandgame.com>
- ▶ Reconvene here at 17:45 and we'll see how far everyone got!