LECTURE 4

STRINGS AND INTEGERS

MCS 260 Fall 2020

Emily Dumas

REMINDERS

- Quiz 1 due today at 6pm Central
  - Excuse requests must be sent to TA before deadline
- Python 3 and editor working?
  - If not, tell me immediately
- Worksheet 2 available, Quiz 2 will be posted soon

STORAGE UNITS

We've discussed the **bit** (b), a binary digit (0 or 1).

A **byte** (B) is a sequence of 8 bits, equivalently, an 8-digit binary number or a 2-digit hex number. It can represent an integer between 0=`0x00` and 255=`0xff`.

A **word** is a longer sequence of bits of a length fixed by the hardware or operating system. Today, a word usually means 16 bits = 2 bytes.

Computers store information as sequences of bytes.

Counting bytes to measure the size of data often leads to large numbers.

Coarser units based on SI prefixes:

- **kilobyte** (KB) = 1,000 bytes
- **megabyte** (MB) = 1,000,000 bytes
- **gigabyte** (GB) = 1,000,000,000 bytes

Based on powers of 2 (IEC system), useful in CS:

- **kibibyte** (KiB) = 2^{10} bytes = 1024 bytes
- **mebibyte** (MiB) = 1024 KiB = 1,048,576 bytes
- **gibibyte** (GiB) = 1024 MiB = 1,073,741,824 bytes

Unfortunate current reality:

- Occasionally, SI abbreviations are used for IEC units; in Windows, "GB" means GiB.
- Very often, IEC units are read aloud using SI names; e.g. write 16GiB and read aloud as "16 gigabytes"

UNICODE

Basic problem: How to turn written language into a sequence of bytes?

Unicode (1991) splits this into two steps:

- Enumerate characters[1] of most[2] written languages; these are **code points**
- Specify a way of **encoding** each code point as a sequence of bytes (not discussed today)

- [1] There are also code points for many non-character entities, such as an indicator of whether the language is read left-to-right or right-to-left.
- [2] Coverage is not perfect and the standard is regularly revised, adding new code points. Unicode 13.0 was released in March 2020.

Every code point has a number (a positive integer between 0 and 0x10ffff=1,114,111).

Code point numbers are always written **`U+`** followed by *hexadecimal digits*.

`U+41`	A
`U+109`	ĉ
`U+1f612`	😒

The first 127 code points, U+0 to U+7F, include all the printable characters on an "en-us" keyboard, numbered exactly as in the older ASCII code (1969).

STRINGS

In Python 3, a **str** is a sequence of code points.

A **string literal** is a way of writing a str in code.

Several syntaxes are supported:

```
'Hello world'  # single quotes
"Hello world"  # double quotes

# multi-line string with triple single quote
'''This is a string
that contains line breaks'''

# multi-line string with triple double quote
"""François: How is MCS 260?
Binali: It's going ok, I guess.
François: [shrugs]"""
```

ESCAPE SEQUENCES

The \ character has special meaning; it begins an **escape sequence**, such as:

- `\n` - the newline character
- `\'` - a single quote
- `\"` - a double quote
- `\\` - a backslash
- `\u0107` - Code point `U+107`
- `\U0001f612` - Code point `U+1f612`

(There is a full list of escape sequences.)

```
>>> print("I \"like\":\n\u0050\u0079\u0074\u0068\u006f\u006e")
I "like":
Python
>>>
```

OPERATIONS ON STRINGS

Most arithmetic operations forbid str operands.

+ is allowed between two strings. It **concatenates** the strings (meaning joins them).

***** is allowed with a string and an int. It concatenates n copies of the string, where n is the int argument.

```
>>> "Hello" + " " + "world!"
'Hello world!'
>>> "Hello" - "llo"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for -: 'str' and 'str'
>>> "Ha" * 4
'HaHaHaHa'
>>> prefix = "Dr. "
>>> fullname = "Ramanujan"
>>> prefix+fullname
'Dr. Ramanujan'
```

LEN AND INDEXING

The built-in **`len()`** can be applied to a string to find the length of the string (a nonnegative int):

```
>>> len("MCS 260")
7
```

A single character from a string **`s`** can be extracted using **`s[i]`** where **`i`** is the 0-based **index**. So 0=first character, 1=second, etc..

```
>>> s = "lorem ipsum"
>>> s[2]
'r'
```

We'll say much more about indexing next time.

INT

When converting from a string, **`int()`** defaults to base 10. But it supports other bases as well. The base is given as the second **argument** of the function.

```
>>> int("1001",2)
9
>>> int("3e",16)
62
```

Notice that integer literal prefixes like **`0b`**, **`0x`**, etc. *must not be present* here. The **`int()`** function works with *just digits*.

However, if a base of 0 is specified, then this signals that the string should be read as a Python literal, i.e. the base is determined by its prefix.

```
>>> int("0b1001",0)
9
>>> int("0x3e",0)
62
>>> int("77",0)
77
```

BITWISE OPERATORS

There are certain operators that only work on ints, and which are based on the bits in the binary expression:

<<	>>	&	|	^
left shift	right shift	bitwise AND	bitwise OR	bitwise XOR

`a << b` moves the bits of **a left** by **b** positions.

`a >> b` moves the bits of **a right** by **b** positions.
(This detroys the lowest **b** bits of **a**.)

```
>>> 9 << 3   # 9 = 0b1001 becomes 0b1001000 = 72
72
>>> 7 << 1   # 7 = 0b111 becomes 0b1110 = 14
14
>>> 9 >> 2   # 9 = 0b1001 becomes 0b10
2
```

Notice `a << b` is equivalent to `a * 2**b`.

Bitwise AND compares corresponding bits, and the output bit is 1 if both input bits are 1:

```
>>> 9 & 5   # 9 = 0b1001,  5 = 0b0101
1
```

	1	0	0	1
	0	1	0	1
AND:	0	0	0	1

Bitwise OR is similar, but the output bit is 1 if at least one of the input bits is 1.

```
>>> 9 | 5   # 9 = 0b1001,  5 = 0b0101
13
```

	1	0	0	1
	0	1	0	1
OR:	1	1	0	1

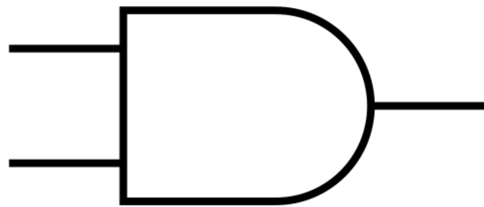
Bitwise XOR makes the output bit 1 if *exactly one* of the input bits is 1.

```
>>> 9 ^ 5   # 9 = 0b1001,  5 = 0b0101
12
```

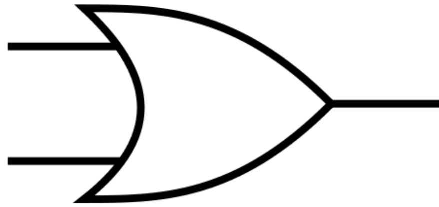
	1	0	0	1
	0	1	0	1
XOR:	1	1	0	0

LOGIC GATES

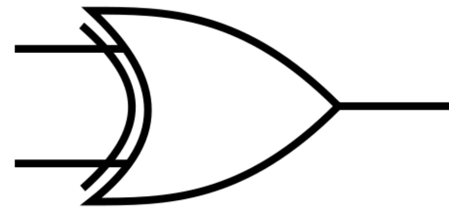
Circuits that perform logic operations on bits, **logic gates**, are fundamental building blocks of computers.



AND

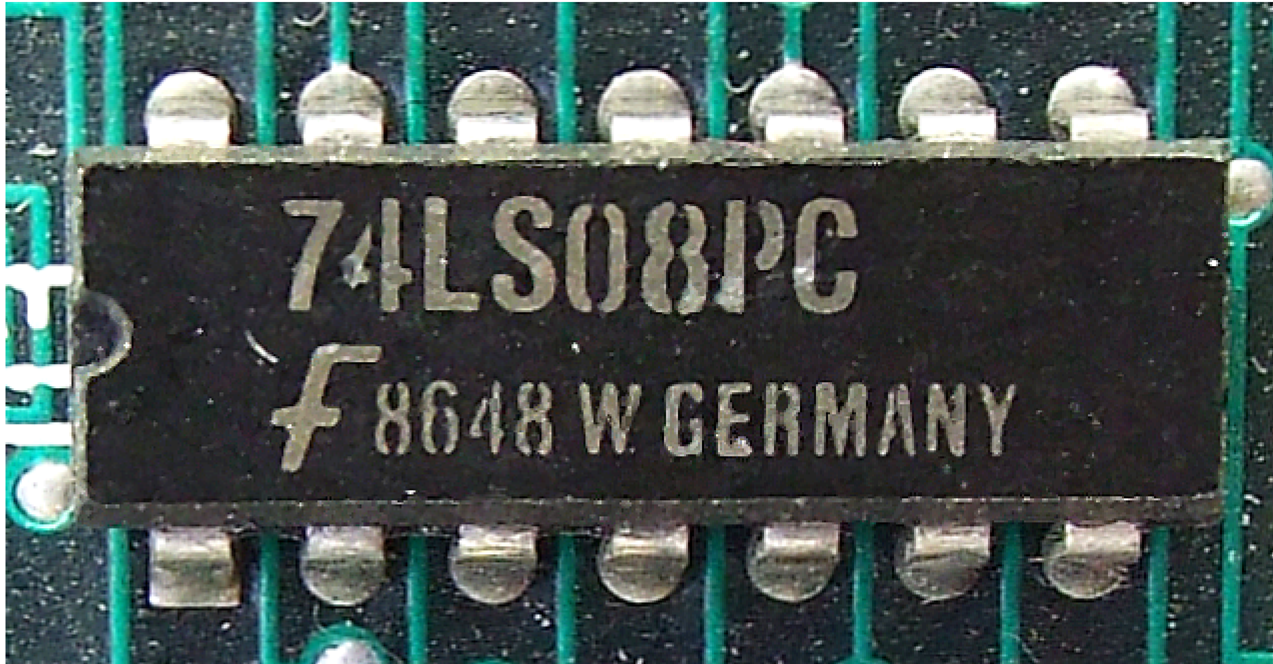


OR



XOR

Thus the Python operators <<,>>,&, | ,^ are especially low-level operations.



74LS08PC photo by Trio3D CC-BY-SA 3.0

This chip (or **integrated circuit** / IC) contains four AND gates built from about 50 transistors. The processor in an iPhone 11 has about 8,500,000,000 transistors.

REFERENCES

- In *Downey*: Strings are discussed in Section 2.6 and Chapter 8
- Bitwise operations in the Python 3 documentation
- The int() feature of converting from strings in other bases is also discussed in the Python 3 documentation.
- Bitwise operations and logic gates are discussed in sections 1.1 and 2.4 of Brookshear & Brylow.

REVISION HISTORY

- 2020-08-31 Typos fixed, explanation of bitwise operators slightly expanded.
- 2020-08-30 Initial publication