LECTURE 7

STRINGS AND INTEGERS

MCS 260 Fall 2021

David Dumas

REMINDERS

- Worksheet 3 available
- Project 1 description posted

BYTES

We've discussed the **bit** (b) or 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`.

Computers store information as sequences of bytes.

UNICODE

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

Unicode (1991) splits this into two steps:

- Make a central directory of characters of most written languages; these are **code points**
- Specify ways to **encode** code points into sequences of bytes (not discussed today)

Every code point has a number (an 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 128 code points, U+0 to U+7F, include all "en-us" keyboard keys, and follow the ASCII code (1969).

STRINGS

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

Several syntaxes are supported for literals:

```
'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.  Too many slides.
François: ¯\_(ツ)_/¯"""
```

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.)

Note \ appears a lot in Windows paths!

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

OPERATIONS ON STRINGS

Most arithmetic operations forbid strings. Exceptions:

- `+` joins strings, e.g. `"cat"+"erpillar"`
- `*` joins a specified number of copies, e.g. `"doo"*6`

```
>>> "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'
```

SEQUENCE STUFF

Reminder: Like lists, strings are sequences.

You can use indexing to get individual characters, slices to get substrings, and `len(...)` to get the length.

STR

Python's `str()` function converts **any** other value to a string, e.g.

```
>>> str(5678)
'5678'
>>> str(5678)[1]
'6'
>>> int(str(5678)[1])
6
```

`str()` is rarely needed, but it does give a way to access decimal digits of an integer individually.

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
```

Integer literal prefixes you'd use in code (`0b`, `0x`, etc.) *must not be present* here. The `int()` function works with *just digits* when you specify the base.

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

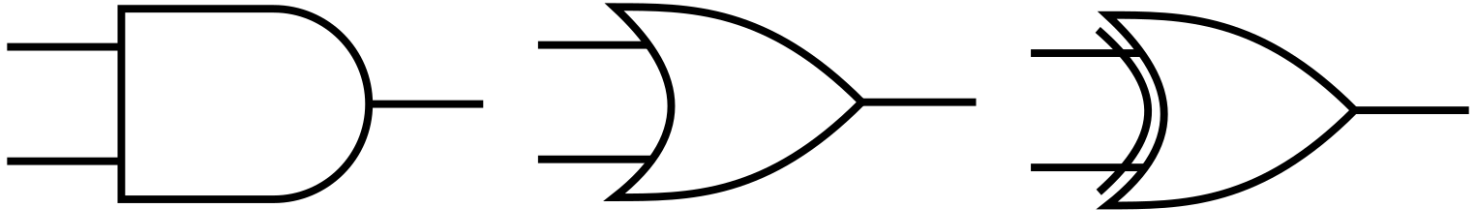
Bitwise XOR makes the output bit 1 if *exactly one* of the input bits is 1.

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

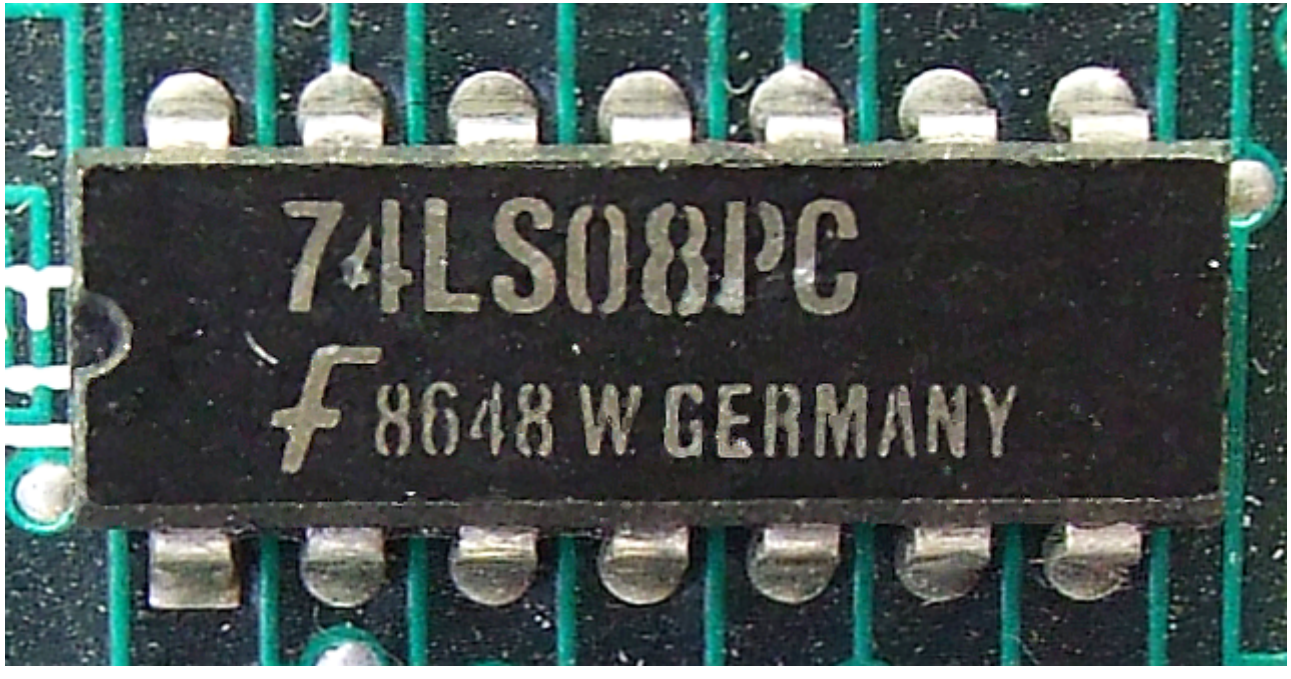
	1	0	0	1
	0	1	0	1
XOR:	1	1	0	0

LOGIC GATES

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



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

- Official Unicode code point charts
- 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

- 2021-09-08 Initial publication
- 2021-09-09 Fix typo