12.010 Computational Methods of Scientific Programming

Tom Herring, tah@mit.edu

Chris Hill, cnh@mit.edu

Lecture 2: Hardware, Boolean algebra, Loops

Topics

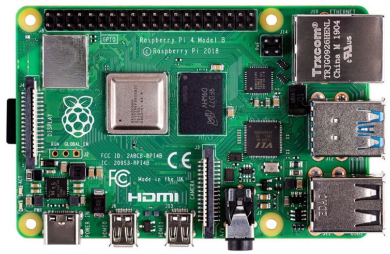
- Review of computer hardware and impact on programming
  - What happens with “Hello World”
- Starting basic concepts: Operators
  - = assignment or binding (this is a critical concept in Python and varies between languages)
- Suites: single blocks of code with header line; Branching and looping
- Boolean Algebra:
  - Logical tests
  - And/or/xor: Truth tables
- Loops:
  - For loops
  - While loops
- Today’s class will get us to looking at the = operator.

Software to hardware

- Most of this class is concerned with tools for writing software, i.e. programming.
- Ultimately programs “execute” on computer hardware
- Hardware imposes some limits of what a program can do, how a program works
- This section looks at some basic hardware concepts that can be useful in understanding what limits it is useful to have in mind when programming.

Modern computers all have similar hardware building blocks

Raspberry Pi 4



Mac Laptop

Summit Supercomputer

Image courtesy of DOE. Image is in the public domain.

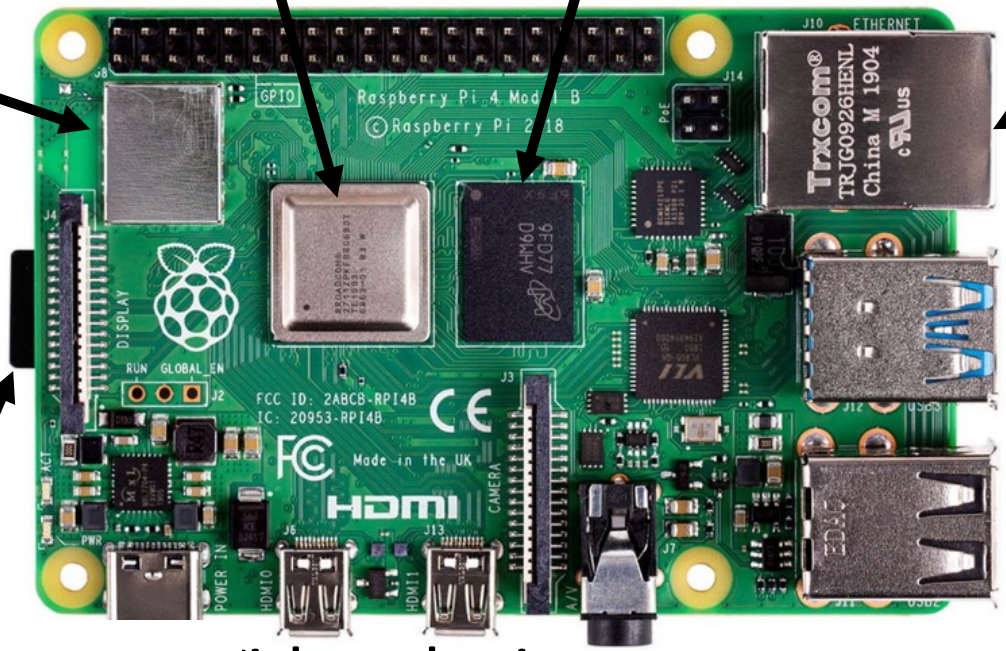
Main hardware pieces are

1. CPU (Intel, AMD, IBM, ARM)
2. dynamic memory (RAM)
3. permanent storage (disk, SSD, memory card)
4. devices (network, graphics/GPU, screen, keyboard)

Wireless network device

CPU

Dynamic Memory (RAM)



Ethernet network device

USB devices

'ideo devices

storage (SD-CARD)

- In programming most of the hardware details are “abstracted” away.
- But physical limits (amount of memory, CPU speed) affect what is possible.

CPU and memory

- All computers have CPU and memory at their heart.
  - A Python programming ultimately is transformed into a sequence of very basic instructions doing a few types of things

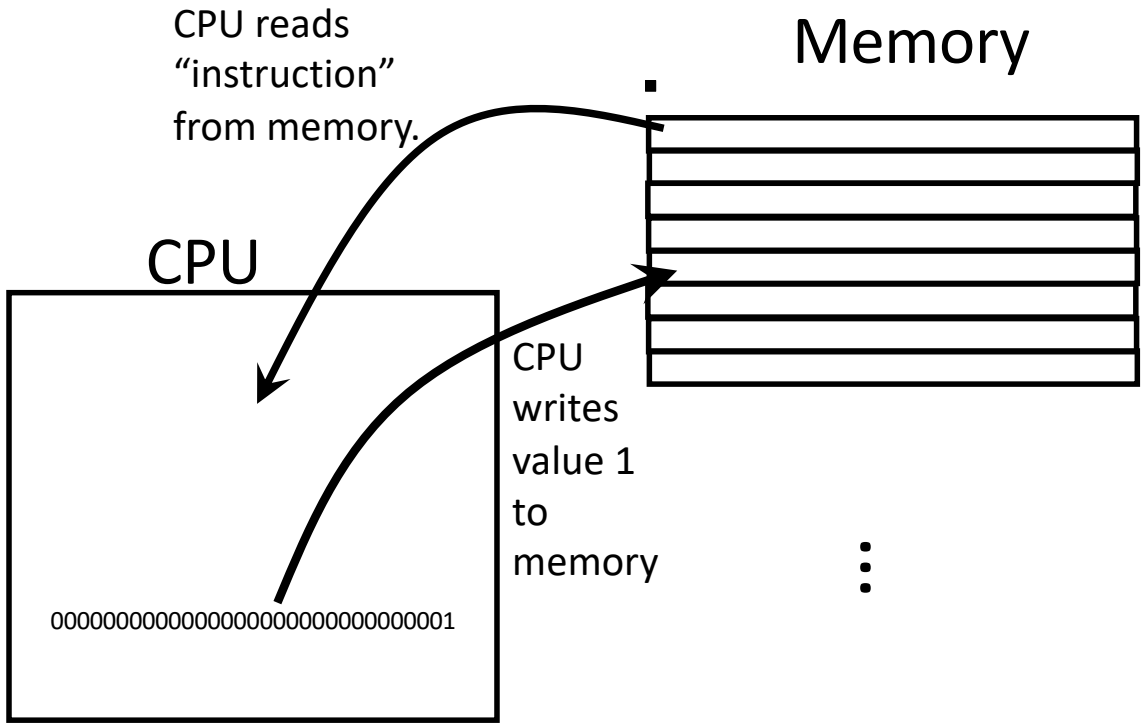
    1. Read some bits (1s and 0s) from memory into a local “register” on a CPU
    2. Combining or comparing different values in different “registers” (i.e. adding, multiplying, testing if less than or bigger than etc…)
       - These computations all work with finite size, binary (sequences of 1 and 0) representations of information
    3. Updating values in ”registers”
    4. Write some bits from a register back to memory.

  - There are variations on each of these steps, but at its heart all a CPU does is lots combinations of 1 – 4 over and over, at very high speed.

An example

Python code

```
[3]: i=1

[4]: i

[4]: 1
```



In response to the Python above

- Python interpreter will write CPU instructions to memory
- CPU will read the instructions
- CPU execute instructions
- Note – 1 here is represented by 32-bits. CPU works in fixed sizes (16-, 32-, 64- bits. In Python, by default, an integer uses 32-bits.

In a modern CPU this is potentially happening billions of times per second.

Memory limits

- Typical laptop had 16GB of memory.
  - 1 byte is 8 bits. 16GB is 16 x 10^9 bytes or 1.28 x 10^{11} bits.
  - 16GB is enough memory for 4 x 10^9 32-bit integers ➔ one matrix with 63,000 x 63,000
  - Because the computer also stores instructions for program and for operating system and data for other programs. The actual limit is less.
  - ➔ in reality, a laptop should be OK to work with matrices up to 2000 x 2000
  - my laptop is OK with adding 40,000 x 40,000
  - with 60,000 x 60,000 - very slow…..

Memory usage and time*

```
[33]: import numpy as np
      import time

[34]: tic = time.time()
      N=4000

[35]: a=np.random.rand(N,N)

[36]: b=np.random.rand(N,N)

[37]: c=a+b

[38]: toc=time.time()

[39]: toc-tic

[39]: 2.0122039318084717
```

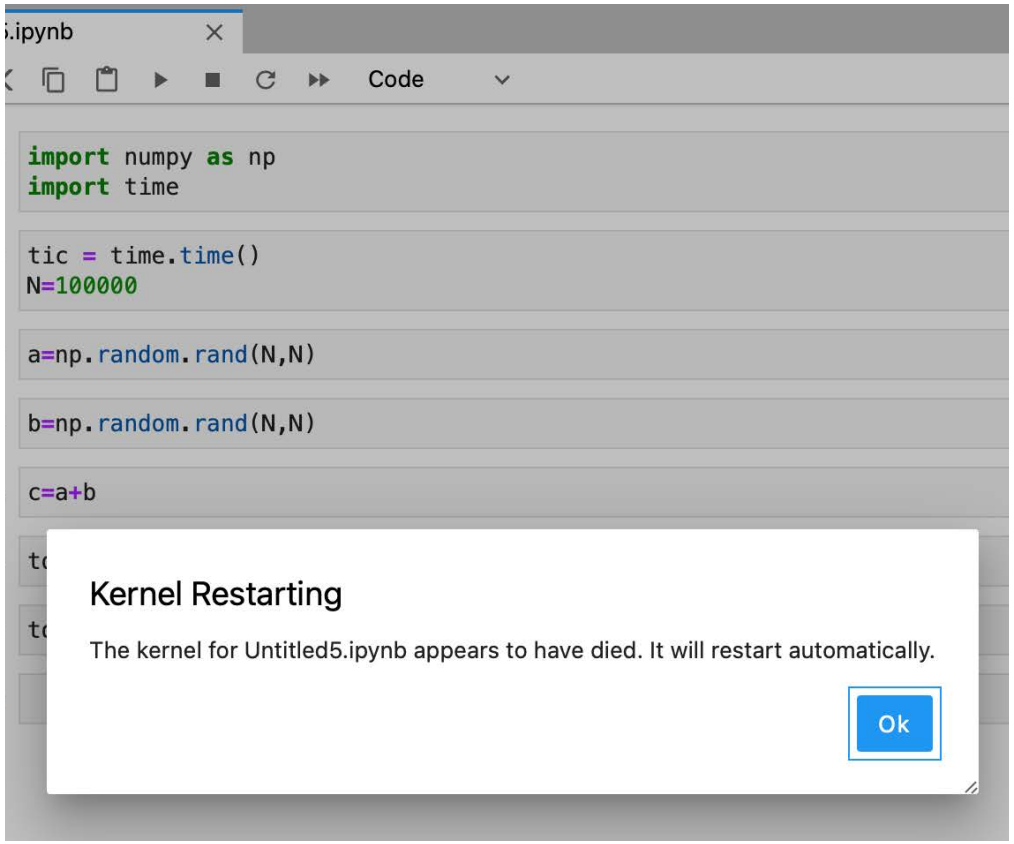
4000 x 4000 => 122MB per matrix

```
[26]: import numpy as np
      import time

[27]: tic = time.time()
      N=40000

[28]: a=np.random.rand(N,N)

[29]: b=np.random.rand(N,N)

[30]: c=a+b

[31]: toc=time.time()

[32]: toc-tic

[32]: 29.106067895889282

[ ]:
```

40000 x 40000 => 11GB per matrix

```
[40]: import numpy as np
      import time

[41]: tic = time.time()
      N=60000

[42]: a=np.random.rand(N,N)

[43]: b=np.random.rand(N,N)

[44]: c=a+b

[45]: toc=time.time()

[46]: toc-tic

[46]: 246.5251431465149

[ ]:
```

60000 x 60000 => 27GB per matrix



100,000 x 100,000 Python kernel dies!

Memory units

- 1 byte – 8-bits
- 1KB – 1024 bytes
- 1Kb – 1024 bits
- 1MB – 1024 x 1024 bytes
- 1GB – 2^{30} bytes
- 1TB – 2^{40} bytes

Variables

1 byte – ASCII character
4 bytes – integer, “single” precision float/real
8 bytes – “long” integer, “double” precision float/real
16 bytes – “quad” precision float/real

For all except ”quad” precision, a CPU will have dedicated circuits (transistors) for working with these sorts of data in “hardware,” i.e., optimized fast computation.

Number representation

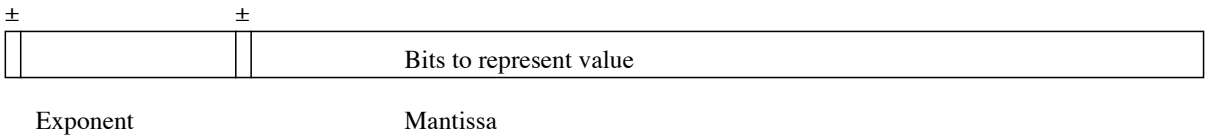
- CPUs can only compute fast on a few specific ways of representing information.
  - Any other representations ultimately use these primitives. This will be slow because the primitives have to combine them in some way to emulate other representaions.
- Key primitives
  - 1-byte characters and Booleans
  - 4-byte integers (32-bit)
  - 8-byte integers
  - 4-byte floating point
  - 8-byte floating point
- For these types a CPU can typical a primitive operation (add/subtract, multiply, compare) in a few CPU clock cycles (typically 1 cycle is 1ns).

Integer numbers

- Integer numbers can be represented exactly (up to the range allowed by the number of bytes)
- A 2-byte integer, unsigned 0-65535, signed ±32767 (sometimes called short)
- A 4-byte integer, unsigned 0-4294967295, signed ±2147483827
- (With a 32-bit address bus, can have 4Gbytes of memory—reason max memory is limited in older computers. Nearly all machines are now 64-bit; still this designation in some software downloads)

Floating point

- Representations vary between machines (often reason binary files can not be shared).



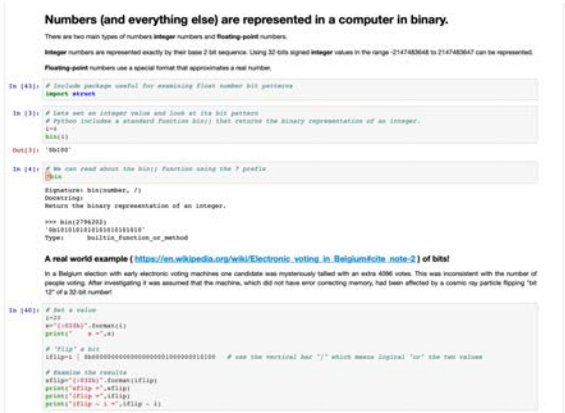
- Precise layout of bits depends on machine and format all formats are (mantissa)*2^(exponent). (Above is not IEEE, exponent is 2s-complement in IEEE) i.e., we think of powers of 10, but computer is powers of 2.
- IEEE: 4-byte floating point is 8 bit exponent, 24 bit mantissa (1 sign bit for each), 7 significant digits, range 10±38

Hands on exercise

- The Python notebook under

https://github.com/christophernhill/fall-2022-12.010/tree/main/looking-at-bits

contains some python code for exploring bit patterns in integers and floating point numbers.



Numbers (and everything else) are represented in a computer in binary.

There are two main types of numbers **integer** numbers and **floating-point** numbers.

Integer numbers are represented exactly by their base 2 bit sequence. Using 32-bits signed **integer** values in the range -2147483648 to 2147483647 can be represented.

Floating-point numbers use a special format that approximates a real number.

```
In [43]: # Include package useful for examining float number bit patterns
         import struct

In [3]: # Lets set an integer value and look at its bit pattern
        # Python includes a standard function bin() that returns the binary representation of an integer.
        i=4
        bin(i)

Out[3]: '0b100'

In [4]: # We can read about the bin() function using the ? prefix
        ?bin

        Signature: bin(number, /)
        Docstring:
        Return the binary representation of an integer.

        >>> bin(2796202)
        '0b1010101011010101010'
        Type:      builtin_function_or_method
```

A real world example (https://en.wikipedia.org/wiki/Electronic_voting_in_Belgium#cite_note-2 **) of bits!**

In a Belgium election with early electronic voting machines one candidate was mysteriously tallied with an extra 4096 votes. This was inconsistent with the number of people voting. After investigating it was assumed that the machine, which did not have error correcting memory, had been affected by a cosmic ray particle flipping "bit 12" of a 32-bit number!

```
In [40]: # Set a value
         i=20
         s="{:032b}".format(i)
         print("   s =",s)

         # "Flip" a bit
         iflip=i | 0b00000000000000000001000000010100  # see the vertical bar "|" which means logical "or" the two values

         # Examine the results
         sflip="{:032b}".format(iflip)
         print("sflip =",sflip)
         print("iflip =",iflip)
         print("iflip - i =",iflip - i)
```

There is Python in the exercise notebook ("bit-int-float.ipynb") that may not be obvious yet. By the end of the class all the features will be explained.
For now we can explore and experiment with some of the code as is.

Operator groups in Python

- Arithmetic operators
- Assignment operators
- Comparison operators
- Logical operators
- Identity operators
- Membership operators
- Bitwise operators

- Other languages have similar types of grouping, but it does vary. Symbols for types of operations can differ and overlap
- Material here based on https://www.w3schools.com/python/python_operators.asp

Arithmetic operators

Operator	Name	Example
+	Addition	x + y
-	Subtraction	x - y
*	Multiplication	x * y
/	Division	x / y
%	Modulus	x % y
**	Exponentiation	x ** y
//	Floor division	x // y

Assignment operators*

- = is the assignment or binding operator. All operators can be used ?= form. Binding can be though of memory location assignment.
- a=b=c=0 # form is also allowed. (operator= form does not work here but there are other subtle effects of op= in terms of binding.
- a,b = 1,2 # acceptable use of assignment statement

Operator	Example	Same As
=	x = 5	x = 5
+=	x += 3	x = x + 3
-=	x -= 3	x = x - 3
*=	x *= 3	x = x * 3
/=	x /= 3	x = x / 3

Summary

- Review of computer hardware and impact on programming
  - What happens with “Hello World”
- Starting basic concepts: Operators
  - = : assignment or binding
- Focus on the difference between assignment and binding.

MIT OpenCourseWare

https://ocw.mit.edu

12.010 Computational Methods of Scientific Programming, Fall 2024