Math 260: Python programming in math

Summer 2020 (Session II)

How python works, and what matters for computation

Under the hood

What happens when you run python code?

source code

```python
23  def dot_prod(x, y):
24      n = len(x)
25      val = 0
26      for j in range(0, n):
27          val += x[j]*y[j]
28
29      return val
```

Python has an **interpreter**:

1) **source code** is converted to **bytecode** comprised of fundamental steps: add, 'load' into memory...

1b) Python 'optimizes' bytecode

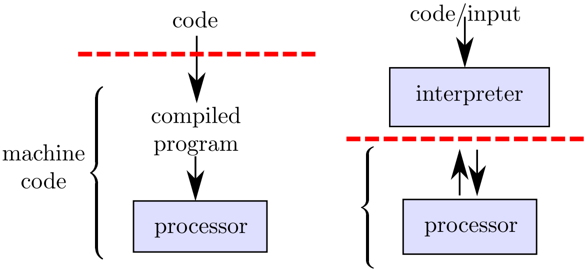
2) The interpreter executes the bytecode....

bytecode

```
# (line 26)
12 SETUP_LOOP              38 (to 52)
14 LOAD_GLOBAL              1 (range)
16 LOAD_CONST               1 (0)
18 LOAD_FAST                2 (n)
20 CALL_FUNCTION            2
22 GET_ITER
24 FOR_ITER                24 (to 50)
26 STORE_FAST               4 (j)
# (line 27)
28 LOAD_FAST                3 (val)
30 LOAD_FAST                0 (x)
32 LOAD_FAST                4 (j)
34 BINARY_SUBSCR
36 LOAD_FAST                1 (y)
38 LOAD_FAST                4 (j)
40 BINARY_SUBSCR
42 BINARY_MULTIPLY
44 INPLACE_ADD
46 STORE_FAST               3 (val)
48 JUMP_ABSOLUTE           24
50 POP_BLOCK
```

Interpreted vs. compiled languages

As it goes from code file to processor instructions, it no longer has 'human-readable' structure (hard to change).



Compiled:

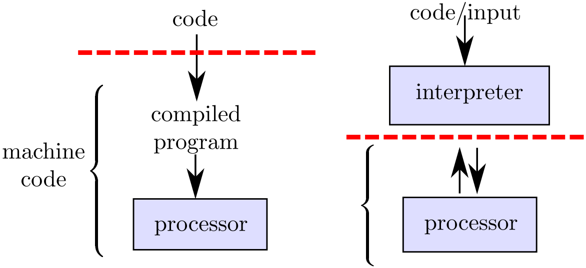
- Code is 'compiled' into a program
  $\rightarrow$ 'machine code' - can be executed
- The whole program must be compiled!
- Cake analogy
- Examples: FORTRAN, C, C++, ...

Interpreted:

- Lines of code $\rightarrow$ interpreter
  a 'virtual' computer that executes
- Interpreter keeps more structure -
  Easier to work with while running
- Examples: python, matlab...

Interpreted vs. compiled languages

As it goes from code file to processor instructions, it no longer has 'human-readable' structure (hard to change).



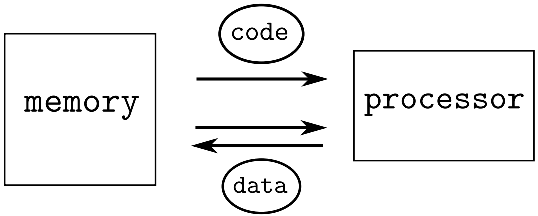
Key point: **compiled code** can be heavily optimized by the compiler (it does not have to keep as much structure from the written code)

Tradeoff: efficiency for flexibility...

Example: `numpy` has fast algorithms coded in C.
The python functions call this (pre-compiled) C-code!

A simple model of a computer

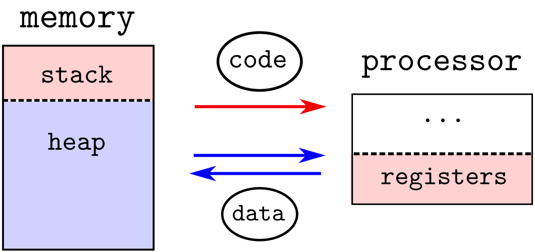
How is code ‘executed’? To answer - we need to know how computers work.
A simple model (to get the point across):



What needs to happen:

- The program and data are loaded into memory
- Data is sent to the processor
- The processor executes instructions
- Data is stored as needed

A simple model of a computer



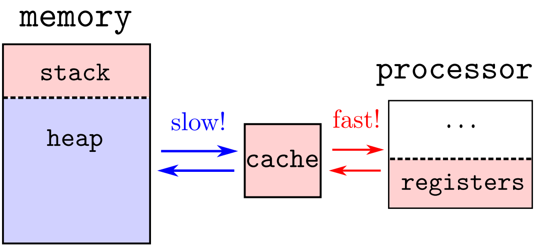
An idealized computer:

- **registers**: memory the processor can use

  Problem: small amount of 'active' space available
- the stack: memory for code, some other items (faster access)
- the heap: memory for data (slow access)

Key question: how to reduce cost of data transfer?

A simple model of a computer



Solution: caching!

- size: register $<$ cache $\ll$ heap
- **cache**: small memory space 'near' registers

  caching: Store frequently used data in the cache

  (keep 'active' data in the cache as long as possible)

So what's the point?

- Caching, memory, etc. handled by python automatically
- But python's ability to optimize **depends on how the code is written**

Why the internal process matters

Example: sum the elements of an $m \times n$ matrix

```
def sum_elements(mat):
    # (asssuming mat is a list of rows, e.g. [[1,2,3],[4,5,6]])
```

- `mat` has entries `mat[j][k]`
- We can sum over j, then k **or** k then j
- Which is faster? Test it!

Quick note: how to time code...

- Use a 'counter' to get the current time
- Save it, then calculate elapsed time at the end
- Good choice (accurate): `time.perf_counter`

```
import time

start = time.perf_counter()
# ... some code goes here...
elapsed = time.perf_counter() - start
print(f"Elapsed time: {elapsed} sec.")
```

Exercise

The timing question, and a bit of review...

a) The following code creates a matrix with entries $a_{jk} = 1/(j+k+1)$ (this is the **Hilbert matrix**):

```
def hilb(n):
    return [[1/(j+k+1) for k in range(n)] for j in range(n)]
```

Rewrite it using two for loops (over j and k).

b) Write a function that sums the elements of a square matrix *by row*:

```
def sum_elements_by_row(mat):
    total = 0
    n = len(mat)
    # ... compute the sum of entries of mat
    return total
```

Example: $A = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$ would be summed as $1+2+3+4$.

c) Now do the same, but *by column* (for A above: $1+3+2+4$.)
Write code in `__main__` to time both and compare, for a large enough n that both take at least a second. Which is faster?

Why the internal process matters

Example: sum the elements of an $m \times n$ matrix

Option 1: rows are 'outer' index

```
def sum_elements(mat):
    sum = 0
    m = len(mat)
    n = len(mat[0])
    for j in range(m):
        for k in range(n):
            sum += mat[j][k]
    return sum
```

Option 2: cols are 'outer' index

```
def sum_elements(mat):
    sum = 0
    m = len(mat)
    n = len(mat[0])
    for k in range(n):
        for j in range(m):
            sum += mat[j][k]
    return sum
```

Choosing the matrix generated by

```
    # an n x n matrix with float elements
    mat = [[1/(j+k+1) for j in range(n)] for k in range(n)]
```

with $n = 3000$ I get:

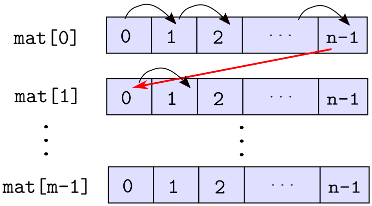
- Option 1: takes ≈ 0.64 sec.
- Option 2: takes ≈ 1.35 sec.

Why the internal process matters

Example: sum the elements of an $m \times n$ matrix

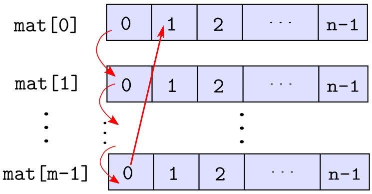
Option 1: rows are 'outer' index

```
def sum_elements(mat):
    total = 0
    m = len(mat)
    n = len(mat[0])
    for j in range(m):
        for k in range(n):
            total  += mat[j][k]
    return total
```



Option 2: cols are 'outer' index

```
def sum_elements(mat):
    total = 0
    m = len(mat)
    n = len(mat[0])
    for k in range(n):
        for j in range(m):
            total  += mat[j][k]
    return total
```



Moving along the array is fast - jumping to the next is slow!
(to be continued...)

Representation of numbers
(and why it matters)

What is a number?

Problem: computers have only a finite amount of data

- **How does the computer approximate a real number?

Questions:

- Is there a largest `int`?

  (Answer: no; python adds space for digits as needed)
- What are the largest and smallest (positive) `floats`?

  Answer: $\approx 10^{-308}$ and $\approx 10^{308}$
- What happens if x goes smaller/larger than these values?

  Answer: (test it!)
- What is the smallest `float` greater than one (i.e. such that `1 + x > 1`)

  Answer: $\epsilon_m \approx 2 \times 10^{-16}$ (**machine epsilon**)

What is a number?

A floating point number is represented in binary in the form

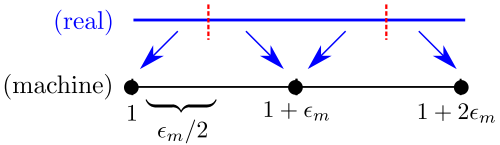
$$1.b_1\, b_2 \cdots b_{52} \times 2^e, \qquad b_k = 0 \text{ or } 1. \tag{F}$$

Example: $7/4$ is $2 \cdot (7/8) = 0.111 \times 2^1$.

- Next number above 1:

$$1 + 2^{-52} = 1 + \epsilon_m$$

- If $|x| < \epsilon_m/2$ then $1 + x$ is rounded to 1



- rounding error for (**??**) has size

$$\frac{1}{2} 2^{-52} \times 2^e$$

- Better to think of the **relative error** ($|x|$ has size 2^e):

$$|x - \texttt{float}(x)| \le \frac{\epsilon_m}{2} |x|$$

What is a number: Example

Rounding error bound:

$$|x - \texttt{float}(x)| \leq \frac{\epsilon_m}{2}|x|$$

Important fact:

Representing real numbers (rounding to `float`), addition/subtraction, multiplication/division all can have an ϵ_m-sized error.
Thus, **every arithmetic operation can introduce an error of size** ϵ_m, which may accumulate through the program.

- In practice, numerical 'equality' is really 'equal up to rounding error':

```
a = (2**(1/2))**2 - 2
print('{:.1e}'.format(a)) #prints 4.4e-16
```

- Expect more error with more computations, e.g.

$$\sum_{n=1}^{10^6} a_n \quad \text{could have error} \sim 10^6\epsilon_m \sim 10^{-10}$$

(although the error is usually better due to coincidental cancellation)

Difference quotients

- Difference quotients can be disastrous....
- Suppose f is computed to an accuracy of ϵ_m. Then

$$\frac{f(y)-f(x)}{y-x} \implies \text{error} \sim \frac{2\epsilon_m}{|y-x|} \gg \text{small}$$

- Dividing 'small' by 'small' **amplifies relative error**.

Simple example:

- Suppose $\sin(\pi/4+10^{-4})$ is computed with an error 10^{-6}. Then

$$\underbrace{\frac{\widetilde{\sin}(\pi/4+10^{-4})-1}{10^{-4}}}_{\text{computed}} = \underbrace{\frac{\sin(\pi/4+10^{-4})-1}{10^{-4}}}_{\text{actual}} + \frac{10^{-6}}{10^{-4}}$$

- Even though the error in sin was 10^{-6}, the error in the quotient is

$$\approx 10^{-2}/\cos(\pi/4) \quad \text{(not small!)}.$$

- Amplified by 10^{-4} (much worse when $y \approx x$).

What is a number: Example

This issue arises naturally in **derivative approximations**.

For instance, consider the **forward difference**

$$f'(x_0) \approx \frac{f(x_0+h)-f(x_0)}{h}.$$

In the world of theory,

$$\lim_{h\to 0} \frac{f(x_0+h)-f(x_0)}{h} = f'(x_0).$$

But for the computer, there are errors in computing f...

- Does the error get better as $h \to 0$?
- Example: take
$$f(x) = \sin x, \qquad x_0 = 1$$
and test for $h = 10^{-k}$ for $k = 1, 2, 3 \cdots, 16$.
- Look at a table/plot of
$$D(h) = \frac{\sin(x_0+h)-\sin(x_0)}{h}, \qquad \text{vs. } h.$$

(see example code)