Python Workshop Series Session 5: *NumPy & Efficient Programming in Python*

Nick Featherstone

Research Computing

Slides: https://github.com/ResearchComputing/Python_Spring_2018

Recall that:

- Python is an *interpreted* language
- Separate program (the interpreter) runs Python code.
- Interpreters execute code “naively.” (line by line)
- Compilers take holistic approach. Interpreters do not.
- Efficiency losses when compared to compiled code.

Compilation vs. Interpretation

Source Code

x =   2*a
x = x + 2*b
x = x + 2*c

Interpreted Program

x =   2*a
x = x + 2*b
x = x + 2*c

3 multiplies; 2 adds

Compiled Program

x = 2*(a + b + c)

1 multiply; 2 adds

Python with Numpy

NumPy provides benefits of compiled language within Python's interpreted framework.

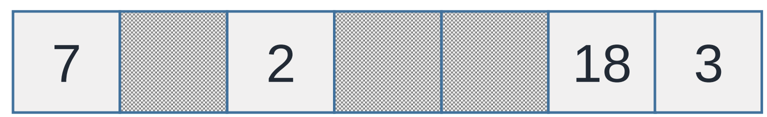
It offers

- Arrays
  (efficient memory access)

- Array methods
  (vectorized loop operations)

A = [7, 2, 18, 3]

memory layout: lists

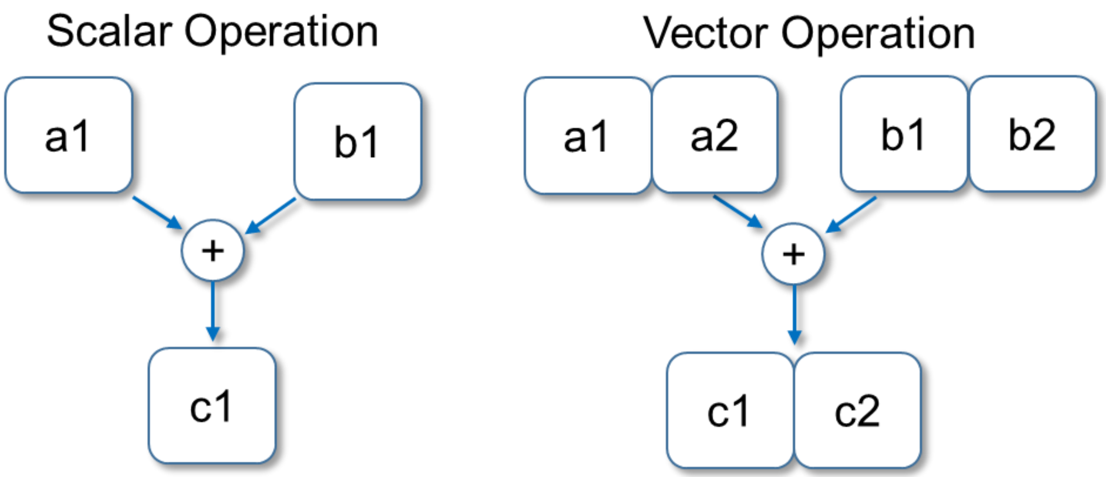


non-contiguous

memory layout: arrays

contiguous

Vectorization?



- Modern processors can perform arithmetic operations on multiple data concurrently
- Think “data parallelism”
- Compiler-enabled

SIMD: Single-Instruction, Multiple Data

-- single instruction (e.g., add, multiply) executed concurrently, by a single process core on multiple pieces of data (e.g., array elements)

Research Computing
UNIVERSITY OF COLORADO BOULDER

The Big Picture

...if you remember nothing else...

Whenever Possible:

- Use NumPy arrays instead of lists
- Use in-place operations
- Use array syntax instead of explicit loops

Getting started with NumPy

- Open **initialization.py**
- Must import the NumPy module first

Common import patterns

```
import numpy

OR

import numpy as np
```

We use this one

NumPy Documentation:

https://docs.scipy.org/doc/numpy/user/index.html

NumPy Array Initialization

```
import numpy as np
my_array = np.init_type (dims, dtype='data type' )
```

my_array : Numy ndarray object (N-dimensional array)

init_type : initialization function

- **zeros** : initialize array with zero values
- **empty** : do not initialize array values

dims : tuple indicating dimensions of the array

e.g., (10) , (10 , 2) , (2 , 8 ,10)

dtype : string variable describing numeric data type

e.g., 'int16', 'int32', 'float16', 'float32', 'float64', 'complex64'

https://docs.scipy.org/doc/numpy/user/basics.types.html

Initializing Arrays with Values

Initialize using values from list

```
list = [ 0, 2, 1, 3]
my_array = np.array (list, dtype='int32' )
```

Initialize using values in [a,b) with integer spacing n

```
my_array = np.arange (a, b, n, dtype='float64' )
```

Initialize using n evenly spaced values in [a, b]

```
my_array = np.linspace (a, b, n, dtype='data type' )
```

Quick Exercises

- Create a 1-D NumPy array with three 16-bit integer elements, initialized to 0.
- Create a 1-D NumPy array with four 64-bit floating-point values initialized to [0, 0.1, 0.2, 0.3] using ***linspace***
- Create a 1-D NumPy array with four 64-bit floating-point values initialized to [1.0 , 0.1 , 9.5 , 11.0] using ***array***

Simple Timing

- In order to talk “efficiency,” we need to time our code.
- Use the **time** function from the **time** module

```
import time
t1 = time.time( )
Test code
t2 = time.time( )
seconds = t2-t1
print(‘Elapsed time: ‘, seconds)
```

Advanced Timing

- Use the cProfile module to profile your code
- https://docs.python.org/3/library/profile.html
- Examine:
  - my_code.py
  - time_my_code.py
- Run **python time_my_code.py**

Use Arrays Instead of Lists

A calculation using NumPy arrays, in conjunction with array syntax, will often complete sooner than one using lists.

Examine:

arrays_vs_lists.py

Avoid Loops When Possible

```
a = np.linspace(…)
b = np.linspace(…)
c = np.zeros(…)
```

Examine:

noloops.py

explicit loop

```
for i in range(n):
    c[i] = a[i]*b[i]
```

not vectorized

Equivalent

array syntax

```
c = a*b
```

vectorized

Exercise

Rewrite **exercise1.py** using

- NumPy arrays instead of lists
- array syntax instead of loops

In-Place Operations

When possible, use in-place operations to avoid unnecessary copies

- a = a + 2 → a += 2
- a = a - 2 → a -= 2
- a = a * 2 → a *= 2
- a = a / 2 → a /= 2

Examine

inplace.py

Array Ordering

- N-D arrays reside in 1-D Memory
- Two different ways of storing arrays

$$A = \begin{bmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{bmatrix}$$

Row-major: stripe row-by-row (C/C++; PYTHON DEFAULT)

Last index is "fastest"

a_{00}	a_{01}	a_{10}	a_{11}

Column-major: stripe column-by-column (Fortran; IDL)

First index is "fastest"

a_{00}	a_{10}	a_{01}	a_{11}

Array Ordering

- We can control the ordering if desired

Examine:
ordering.py

Row-major: stripe row-by-row (C/C++; PYTHON DEFAULT)

Last index is “fastest”

a_{00}	a_{01}	a_{10}	a_{11}

Column-major: stripe column-by-column (Fortran; IDL)

First index is “fastest”

a_{00}	a_{10}	a_{01}	a_{11}

Array Ordering: Why Care?

- Sometimes, you REALLY need a loop.
- The innermost loop should correspond to the fastest array index.

Examine:

access_patterns.py

Row-Major

```
for j in range(m)
        for k in range(n):
                a+=b[ j ][ k ]
```

Column-Major

```
for k in range(n)
        for j in range(m):
                a+=b[ j ][ k ]
```

I/O with Numpy Arrays

- Writing/reading numpy arrays to/from a file is easy…

Examine:

numpy_io.py

- Arrays are ALWAYS written in Row-Major Order
- Not portable (Endianness is machine-specific)
- Intel processors are little-endian
- Better to use standard like HDF5

Recall: Binary

- Base 2 numbering system
- Each digit referred to as a bit
- Your computer does math in binary
- Modern computers work in bytes: groups of 8 bits

Examples:

$000 = 0x2^2 + 0x2^1 + 0x2^0 = 0$

$001 = 0x2^2 + 0x2^1 + 1x2^0 = 1$

$110 = 1x2^2 + 1x2^1 + 0x2^0 = 6$

$101 = 1x2^2 + 0x2^1 + 1x2^0 = 5$

Endianness

- Computers store numbers in bytes (8 bits)

- Most computers store their bytes in one of two different ways:
  - Most significant (i.e., largest 2^n’s) byte first (Big Endian)
  - Least significant (i.e., smallest 2^n’s) byte first (Little Endian)

Example: 16-bit Binary

769 = 0000001100000001

00000011	00000001

Big Endian Ordering

00000001	00000011

Little Endian Ordering

What is your machine? Run endian_check.py.
Command line: xxd -b 769.dat