Efficient Serial Programming with NumPy

Nick Featherstone
CU Research Computing

[Web Link to These Slides]

Performance & Python

Python is an interpreted language

- No binary executable is created
- Interpreter executes source code line-by-line
- Instructions are executed *naively*
    - exactly as you wrote them
    - In the order you wrote them
    - no inherent vectorization

Compiled languages are different

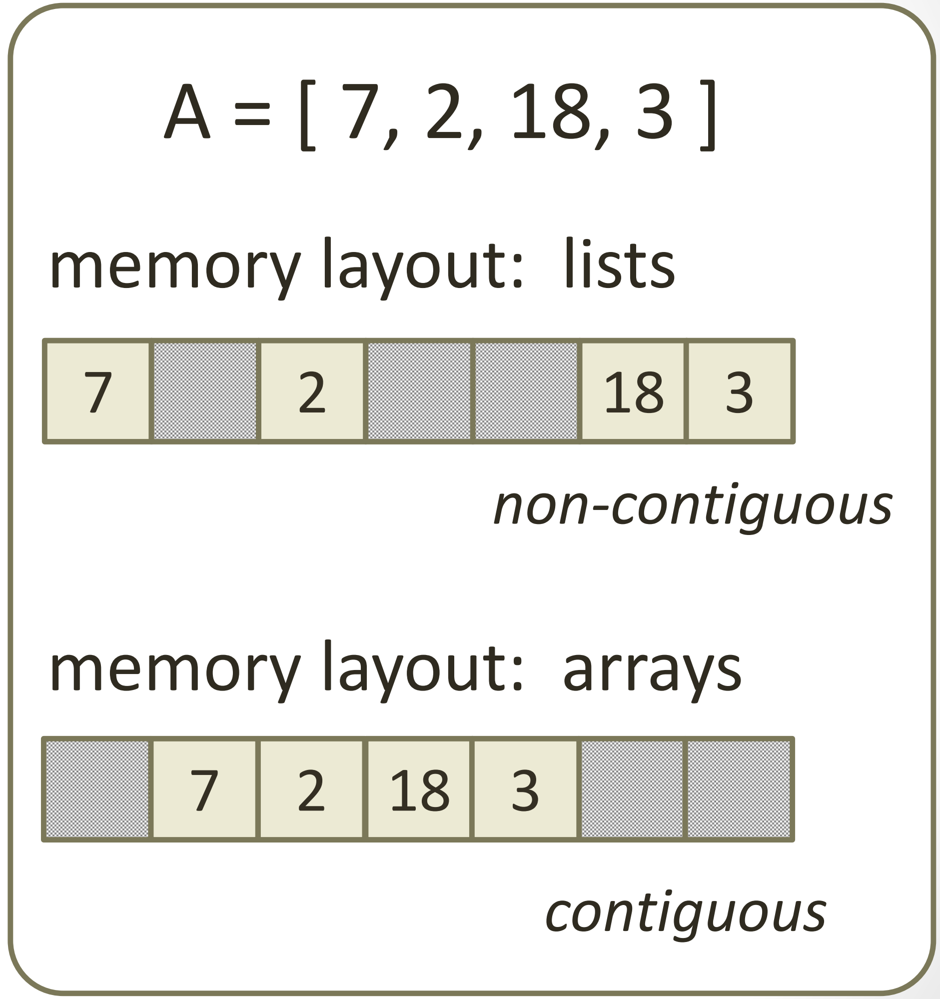
- A binary executable is created
- Instructions are executed *holistically*
    - Same outcome as naive approach
    - Compiler-assisted optimization & vectorization
    - Implementation differs between compilers

Python with Numpy

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

It offers

- Arrays
  (efficient memory layout)
- Array methods
  (vectorized loop operations)



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 in order to use its features

Common import patterns

```
import numpy

OR

import numpy as np
```

We use this

Open this file:

Parallelization Workshop /

Day3-Parallel_Python /

session1_numpy /

examples /

initialization.py

NumPy Documentation:

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

NumPy ndarray Intialization Pattern

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

init_type : describes whether to initialize array to zero or not
- zeros -> initialize array with zero values
- empty -> do not initialize array values

dims : tuple with dimensions (nx, [ny, nz, …]) of the array
e.g., (10), (10,2), (2,8,10)

dtype : string variable describing the type of variable
e.g., 'int16', 'int32', 'float16', 'float32', 'float64', 'complex64'

more on data types: 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='data type' )
```

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

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

Initialize using n evenly space values on [a, b]

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

Quick Exercises

- Create a 1-D NumPy array with 3 16-bit integer elements, initialized to 0.

- Create a 1-D NumPy array with 4 64-bit floating-point values initialized to [0, 0.1, 0.2, 0.3] using *linspace*

- Create a 1-D NumPy array with 4 64-bit floating-point values initialized to [1.0 , 0.1 , 9.5 , 11.0] using *array*

Use Arrays Instead of Lists

A calculation using NumPy arrays, in conjunction with array syntax, will usually be faster than one using lists.

Open this file:

Parallelization Workshop /

Day3-Parallel_Python /

session1_numpy /

examples /

arrays_vs_lists.py

Avoid Loops When Possible

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

Open this file:

Parallelization Workshop /

Day3-Parallel_Python /

session1_numpy /

examples /

noloops.py

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

Equivalent

```
c = a*b
```

explicit loop
not vectorized

array syntax
vectorized!

Exercise1

Rewrite this program using

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

Open this file:

Parallelization Workshop /

Day3-Parallel_Python /

session1_numpy /

exercises /

exercise1.py

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

Open this file:

Parallelization Workshop /

Day3-Parallel_Python /

session1_numpy /

examples /

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)

First index is "fastest"

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

Array Ordering

- We can control the ordering if desired

Open this file:
Parallelization Workshop / Day3-Parallel_Python / session1_numpy /examples / **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)

First index is "fastest"

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

Array Ordering: Why Care?

- Sometimes, you REALLY do have to write a loop

- The innermost loop should correspond to the fastest array index

Open this file:
Parallelization Workshop /
Day3-Parallel_Python /
session1_numpy /
examples /
access_patterns.py

Row-Major

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

Column-Major

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

I/O with Numpy Arrays

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

Open this file:

Parallelization Workshop /

Day3-Parallel_Python /

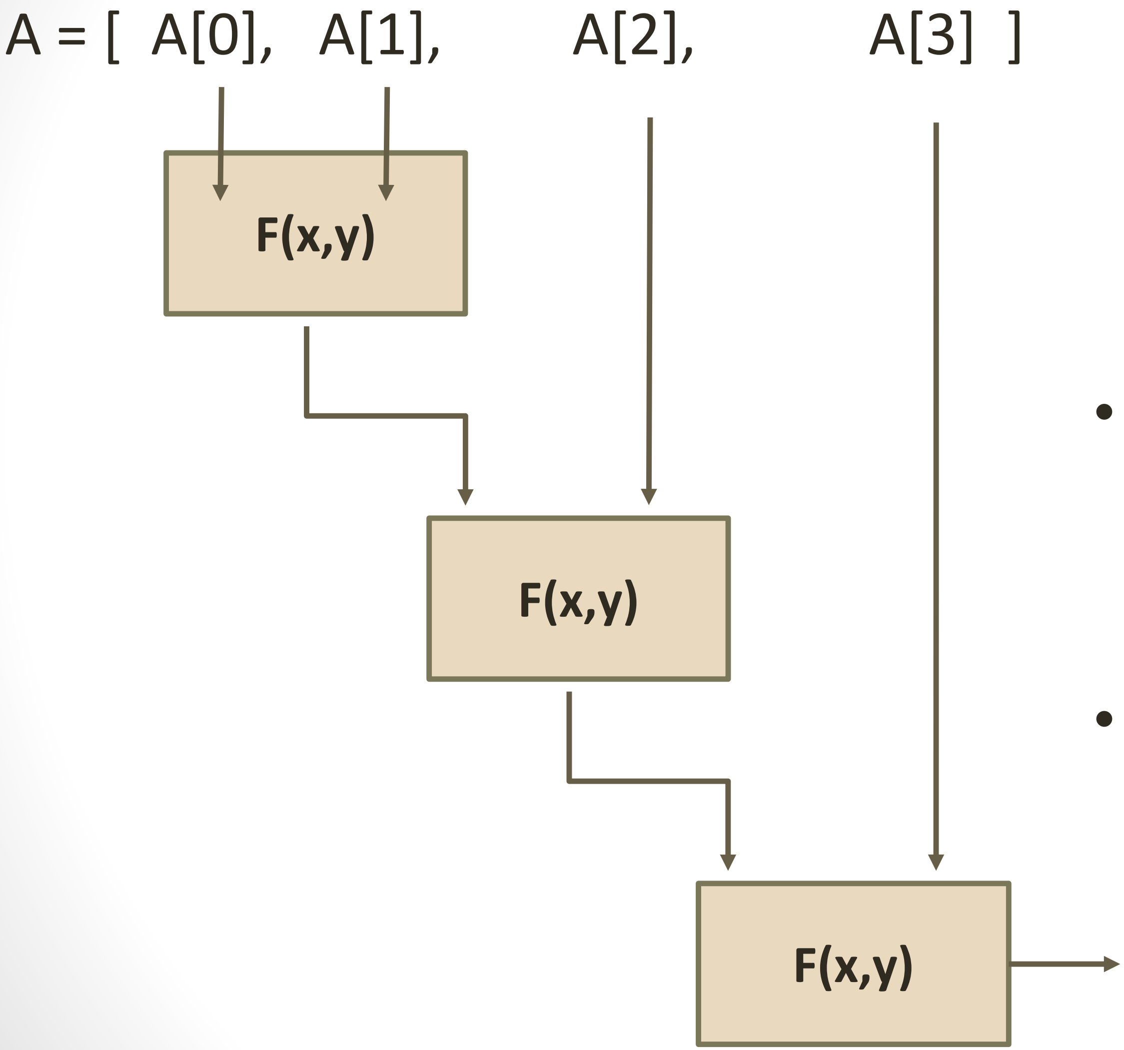
session1_numpy /

examples /

numpy_io.py

- NOTE: Arrays are ALWAYS written in Python/C-style Row-Major Order

Reduction



- Sequential function evaluation using results from previous evaluations in tandem with new inputs
- Think:
  - Adding numbers
  - Computing factorials
- Common in parallel applications

Scalar Result

Function Mapping:

- Useful shorthand for performing function evaluation on each element contained within a list of numbers
- Returns a list
- Usage:
  results = map(function_name, list_name)

Open this file:

Parallelization Workshop / Day3-Parallel_Python /

session1_numpy / examples /

map_reduce.py

Exercise 2

Rewrite this program using

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

Open this file:

Parallelization Workshop /

Day3-Parallel_Python /

session1_numpy /

exercises /

exercise2.py