CS2204

Program Design and Data Structures for Scientific Computing

Announcements

- Project #2 is posted
  - We will be creating another class that allows us to represent & manipulate DNA strands
  - We will use a list as our underlying storage container
  - Our goal is to have a faster/smaller representation by utilizing the mutability of lists (as compared to the immutability of strings)
    - You should use mutation operations on your list whenever possible
    - Avoid operations that create new lists

Matrix Programming

Introduction

By Richard T. Guy

Studying patients with Babbage's Syndrome

How effective are available treatments (A, B, & C)?

	A	B	C
John	**2.5**	**3.5**	**3.0**
Mary	**3.0**	**1.5**	**3.0**
Zura	**2.5**	**2.0**	**5.5**

How similar are patients' responses?

Can we use similarity to recommend treatments?

We can answer these questions with matrix algebra.

How to implement them in software?

Option 1: write loops

- Makes programs many times longer than the corresponding mathematics
- And it's hard code to debug...
- ...and tune

Option 2: use libraries written in low-level, high-performance languages like Fortran and C

– Someone else has written, debugged, and tuned all the loops

– But the interface is... awkward

SUBROUTINE CAXPY(N,CA,CX,INCX,CY,INCY)

complex

constant
times vector
plus vector

number of
elements

vectors

spacing

Option 3: use a high-level language like MATLAB

Or a library like Python's NumPy

- Present a *data-parallel* programming model
  - Operate on entire arrays at once
  - No loops!
- Hide details of optimizations
  - Particularly differences between machines
- All provide basically the same features
  - Often wrappers around the same underlying libraries

software carpentry

NumPy (http://numpy.scipy.org)

Provides MATLAB-style arrays for Python

And many other things

A *data parallel* programming model

- Write **x*A*x.T** to calculate xAx^T
- The computer takes care of the loops

All encapsulated in special objects called *arrays*

Create an array from a list

```
>>> import numpy
>>> vals = [3, 5, 7]
>>> arr = numpy.array(vals)
>>> arr
array([3, 5, 7])
```

Alternatively…

```
>>> from numpy import *
>>> vals = [3, 5, 7]
>>> arr = array(vals)
>>> arr
array([3, 5, 7])
```

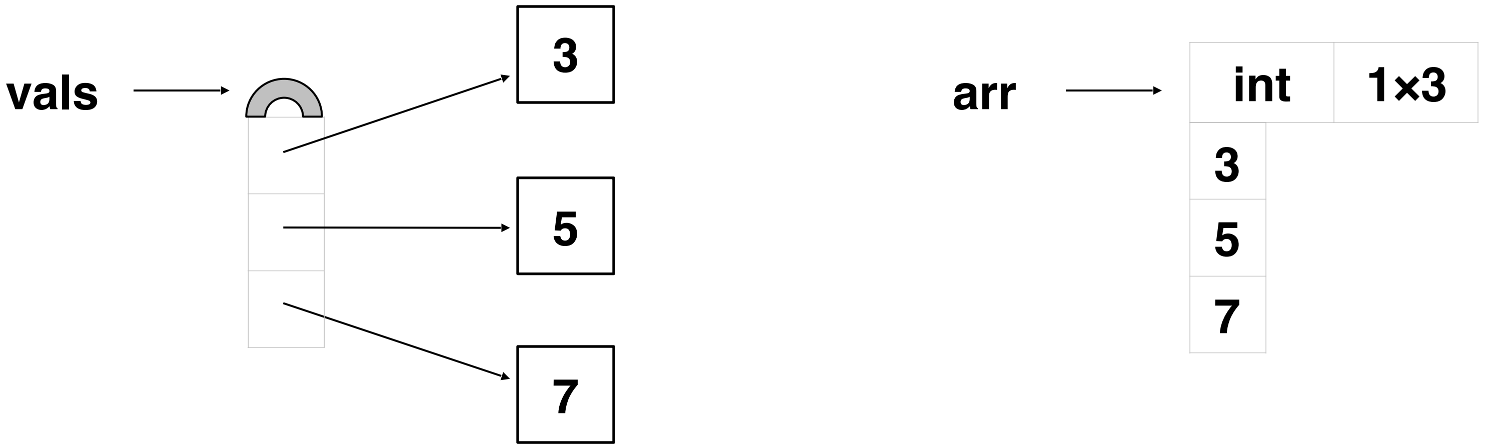
Arrays are *homogeneous*

– I.e., all values have the same type

Allows values to be packed together

– Saves memory

– Faster to process



So what does this do?

```
>>> arr = numpy.array([1, 2.3])
```

So what does this do?

```
>>> arr = numpy.array([1, 2.3])
>>> arr
array([1., 2.3])
```

A float, not an int

If we give NumPy initial values of different types…

…it finds the most general type---in this case, float---and uses that.

You can specify a type at creation time:

```
>>> array([1, 2, 3, 4], dtype=float32)
array([ 1., 2., 3., 4.])
```

software carpentry

You can also specify the data type later

```
>>> a = array([1, 2, 3, 4], dtype=float32)
>>> a.astype(int)
array([ 1, 2, 3, 4])
>>> #the above returns a new array
```

You can also specify the data type later

```
>>> a = array([1, 2, 3, 4], dtype=float32)
>>> a.astype(int)
array([ 1, 2, 3, 4])
>>> #the above returns a new array
>>> a.dtype = int
>>> a
array([1065353216, 1073741824, 1077936128,
   1082130432])
>>> #the above changed the type field, causing the bits
   to be interpreted differently
```

Basic data types are:

bool	**uint[8,16,32,64]**
int	**float**
int8	**float[16,32,64,128]**
int16	**complex**
int32	**complex[64,128]**
int64	

Many other ways to create arrays

```
>>> z = numpy.zeros((2, 3))
>>> z
array([[0., 0., 0.],
       [0., 0., 0.]])
```

- Type is **float** unless something else specified
- The 'zeros' function takes a tuple specifying array dimensions

Many other ways to create arrays

```
>>> z = numpy.zeros((2, 3))
>>> z
array([[0., 0., 0.],
       [0., 0., 0.]])
```

- Type is **float** unless something else specified
- The 'zeros' function takes a tuple specifying array dimensions

What do these do?

```
>>> block = numpy.ones((4, 5))
>>> mystery = numpy.identity(4)
```

Can create arrays without filling in values

```
>>> x = numpy.empty((2, 2))
>>> x
array([[3.82265e-297, 4.94944e+173],
       [1.93390e-309, 1.00000e+000]])
```

"Values" will be whatever bits were in memory

Should not be used without being initialized

When is this useful?

As with everything, assigning creates alias: does *not* copy data

```
>>> first = numpy.ones((2, 2))
>>> first
array([[1., 1.],
       [1., 1.]])
>>> second = first
>>> second[0, 0] = 9
>>> first
array([[9., 1.],
       [1., 1.]])
```

As with everything, assigning creates alias: does *not* copy data

```
>>> first = numpy.ones((2, 2))
>>> first
array([[1., 1.],
       [1., 1.]])
>>> second = first
>>> second[0, 0] = 9
>>> first
array([[9., 1.],
       [1., 1.]])
```

Not : **second[0][0]**

Use the **array.copy** method to make a copy

```
>>> first
array([[1., 1.],
       [1., 1.]])
>>> second = first.copy()
>>> second[0, 0] = 9
>>> first
array([[1., 1.],
       [1., 1.]])
```

Arrays also have properties

```
>>> first
array([[1., 1.],
       [1., 1.]])
>>> first.shape
(2, 2)
>>> block = numpy.zeros((4, 7, 3))
>>> block.shape
(4, 7, 3)
```

Arrays also have properties

```
>>> first
array([[1., 1.],
       [1., 1.]])
>>> first.shape
(2, 2)
>>> block = numpy.zeros((4, 7, 3))
>>> block.shape
(4, 7, 3)
```

Not a method call

Arrays also have properties

```
>>> first
array([[1., 1.],
       [1., 1.]])
>>> first.shape
(2, 2)
>>> block = numpy.zeros((4, 7, 3))
>>> block.shape
(4, 7, 3)
```

Not a method call

Consistent

array.size is the total number of elements

```
>>> first.size
4
>>> block.size
84
```

4 ← 2×2

84 ← 4×7×3

Reverse on all axes with **array.transpose**

```
>>> first = numpy.array([[1, 2, 3],
                         [4, 5, 6]])
>>> first.transpose()
array([[1, 4],
       [2, 5],
       [3, 6]])
>>> first
array([[1, 2, 3],
       [4, 5, 6]])
```

this creates an alias that appears to have the values stored differently; does not actually copy all the data

Flatten arrays using **array.ravel**

```
>>> first = numpy.zeros((2, 2, 2))
>>> second = first.ravel()
>>> second.shape
(8,)
```

this creates a one-dimensional alias for the original data

Think about the 2×4 array A:

```
>>> A
array([[1, 2, 3, 4],
       [5, 6, 7, 8]])
```

'A' looks 2-dimensional

But computer memory is 1-dimensional

Must decide how to lay out values

Row-major order concatenates the rows

- Used by C and Python

1	2	3	4
5	6	7	8

Logical

1	2	3	4	5	6	7	8

Physical

Column-major order concatenates the columns

- Used by Fortran and MATLAB

1	2	3	4
5	6	7	8

Logical

1	5	2	6	3	7	4	8

Physical

No difference in usability or performance[1]…

…but causes headaches when passing data from one language to another

(Just like 0-based vs. 1-based indexing)

1: Performance can be affected if you write your own loops over large matrices; you want to process the matrix in physical order if possible

Can *reshape* arrays in many other ways

```
>>> first = numpy.array([1, 2, 3, 4, 5, 6])
>>> first.shape
(6,)
>>> second = first.reshape(2, 3)
>>> second
array([[1, 2, 3],
       [4, 5, 6]])
```

Tuple with 1 element

Not packed into a tuple

Also aliases the data

New shape must have same size as old

```
>>> first = numpy.zeros((2, 2))
>>> first.reshape(3, 3)
ValueError: total size of new array must
be unchanged
```

Cannot possibly work because it is just creating an alias for the existing data

Change physical size using **array.resize**

```
>>> block
array([[ 10,  20,  30],
       [110, 120, 130],
       [210, 220, 230]])
>>> block.resize(2, 2) #may or may not work, depends on
                                python version
>>> block
array([[ 10,  20],
       [110, 120]])
```

This did not work for me; see next slide…

Change physical size using **resize**

```
>>> block
array([[ 10,  20,  30],
       [110, 120, 130],
       [210, 220, 230]])
>>>resize(block,(2, 2))
array([[ 10,  20],
       [30, 110]])
```

Review:

– Arrays are blocks of homogeneous data

– Most operations create aliases

– Can be reshaped (size remains the same)

– Or resized