UPPSALA
UNIVERSITET

High Performance Computing: Speed Optimisation Using NumPy, Cython, MPI And GPU Acceleration

2017-05-17

Advanced Scientific Programming with Python

Why NumPy?

```
import numpy as np

a = np.random.random((1000000))
b = list(a)
%timeit a.sum()
%timeit sum(b)
```

```
In [4]: %timeit a.sum()
1000 loops, best of 3: 415 µs per loop

In [5]: %timeit sum(b)
10 loops, best of 3: 41.3 ms per loop

In [6]: 41.3/0.415
Out[2]: 99.51807228915662
```

socrative
UU1

How much faster is the NumPy version?

A - About 2x

B - About 5x

C - About 25x

D - About 100x

E - NumPy is slower

What Is NumPy?

- The `numpy` package is used in almost all numerical computation using Python.
- It provide high-performance vector, matrix and higher-dimensional data structures for Python.
- It is implemented in C and Fortran so when calculations are vectorized (formulated with vectors and matrices), **performance is very good**.

Fancy Indexing

- NumPy offers more indexing facilities than regular Python sequences.
- In addition to indexing by integers and slices, arrays can be indexed by arrays of integers and arrays of booleans.

```
>>> a = np.arange(12)**2                       # the first 12 square numbers
>>> i = np.array( [ 1,1,3,8,5 ] )              # an array of indices
>>> a[i]                                       # the elements of a at the positions i
array([ 1,  1,  9, 64, 25])
>>>
>>> j = np.array( [ [ 3, 4], [ 9, 7 ] ] )      # a bidimensional array of indices
>>> a[j]                                       # the same shape as j
array([[ 9, 16],
       [81, 49]])
```

- You can also use indexing with arrays as a target to assign to:

```
>>> a = np.arange(5)
>>> a
array([0, 1, 2, 3, 4])
>>> a[[1,3,4]] = 0
>>> a
array([0, 0, 2, 0, 0])
```

Fancy Indexing

- When we index arrays with arrays of (integer) indices we are providing the list of indices to pick.
- With boolean indices the approach is different; we explicitly choose which items in the array we want and which ones we don't.
- The most natural way one can think of for boolean indexing is to use boolean arrays that have the same shape as the original array:

```
>>> a = np.arange(12).reshape(3,4)
>>> b = a > 4
>>> b                                     # b is a boolean with a's shape
array([[False, False, False, False],
       [False,  True,  True,  True],
       [ True,  True,  True,  True]], dtype=bool)
>>> a[b]                                  # 1d array with the selected elements
array([ 5,  6,  7,  8,  9, 10, 11])
```

Fancy Indexing

- Fancy indexing is the most general selection method, but it is also the slowest.

```
# Let's create an array with a large number of rows.
# We will select slices of this array along the first dimension.
n, d = 100000, 100
a = np.random.random_sample((n, d)); aid = id(a)

# Let's select one every ten rows, using two different methods
# (array view and fancy indexing).
b1 = a[::10]
b2 = a[np.arange(0, n, 10)]
np.array_equal(b1, b2)
True

# Let's compare the performance of both methods.
%timeit a[::10]
1000000 loops, best of 3: 804 ns per loop
%timeit a[np.arange(0, n, 10)]
100 loops, best of 3: 14.1 ms per loop
```

- Fancy indexing is several orders of magnitude slower as it involves copying a large array.

Fancy Indexing

```
>>> a = np.arange(5)
>>> a[[0,0,2]]=[1,2,3]
>>> a
```

```
>>> a
array([2, 1, 3, 3, 4])
```

socrative
UU1

What's the value of a[0]?

```
>>> a = np.arange(5)
>>> a[[0,0,2]]+=1
>>> a
array([1, 1, 3, 3, 4])
```

socrative
UU1

And in this case what's a[0]?

Views

- A view is simply another way of viewing the data of the array.
- The data of both objects are shared.
- You can create views by selecting a slice of the original array or by changing the data type.
- Slice views are the most common.
- The rule of thumb for creating a slice view is that the viewed elements can be addressed with offsets, strides, and counts in the original array. For example:

Slice Views

- Are the most common view.
- The viewed elements must be able to be addressed with offsets, strides, and counts in the original array.

```
>>> a = numpy.arange(10)
>>> a
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> v1 = a[1:2]
>>> v1
array([1])
>>> a[1] = 2
>>> v1
array([2])
>>> v2 = a[1::3]
>>> v2
array([2, 4, 7])
>>> a[7] = 10
>>> v2
array([ 2,  4, 10])
```

- Any selection which involves only **`slices`** (e.g. **`0:10`**, **`::-1`**, etc…) returns a view.

Data Type Views

- Sometimes you want to inspect memory with a different data type
- This is when data type views come in

```
In [12]: b = numpy.arange(10, dtype='int16')
In [13]: b
Out[13]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9], dtype=int16)
In [14]: v3 = b.view('int32')
In [15]: v3
Out[15]: array([ 65536, 196610, 327684, 458758, 589832], dtype=int32)
In [16]: v3 += 1
In [17]: b
Out[17]: array([1, 1, 3, 3, 5, 5, 7, 7, 9, 9], dtype=int16)
In [18]: v4 = b.view('int8')
In [19]: v4
Out[19]: array([1, 0, 1, 0, 3, 0, 3, 0, 5, 0, 5, 0, 7, 0, 7, 0, 9, 0, 9, 0],
               dtype=int8)
```

- You get the raw byte stream casted to whatever data type you selected

Data Ownership

- Now you see that multiple arrays can modify the same data
- Who owns the data?

```
In [20]: b.flags
Out[20]:
  C_CONTIGUOUS : True
  F_CONTIGUOUS : True
  OWNDATA : True
  WRITEABLE : True
  ALIGNED : True
  UPDATEIFCOPY : False

In [21]: v4.flags
Out[21]:
  C_CONTIGUOUS : True
  F_CONTIGUOUS : True
  OWNDATA : False
  WRITEABLE : True
  ALIGNED : True
  UPDATEIFCOPY : False
```

View And Data Ownership

```
In [26]: a = numpy.array([1])

In [27]: b = a[:]

In [28]: b[0] = 0

In [29]: a[0] == 0

Out[29]: True
```

socrative
UU1

What is the result of the code?

```
In [36]: a = numpy.random.random((3))

In [37]: b = a[::-1]

In [38]: b[0] = 1

In [39]: b[0] == a[-1]
Out[39]: True
```

And now?

View And Data Ownership

```
In [40]: a = numpy.random.random((3))

In [41]: b = a

In [42]: b[0] = 1

In [43]: b[0] == a[0]
Out[43]: True
```

socrative
UU1

And in this one?

```
In [44]: a = numpy.random.random((3))

In [45]: b = a[[0,1,2]]

In [46]: b[0] = 1

In [47]: b[0] == a[0]
Out[47]: False
```

And finally?

Some Tricky Cases

Both snippets on the right seem to do the same thing but they don't.

```
>>> a = numpy.arange(10)
>>> a[[1,2]] = 100
>>> a
array([  0, 100, 100,   3,   4,   5,   6,   7,   8,   9])
```

```
>>> a = numpy.arange(10)
>>> c1 = a[[1,2]]
>>> c1[:] = 100
>>> a
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> c1
array([100, 100])
```

Some Tricky Cases

```
>>> a = numpy.arange(12).reshape(3,4)
>>> ifancy = [0,2]
>>> islice = slice(0,3,2)
>>> a[islice, :][:, ifancy] = 100
>>> a
array([[100,   1, 100,   3],
       [  4,   5,   6,   7],
       [100,   9, 100,  11]])
```

socrative UU1

Is a[0,0] == 100?

```
>>> a = numpy.arange(12).reshape(3,4)
>>> ifancy = [0,2]
>>> islice = slice(0,3,2)
# note that ifancy and islice
# are interchanged here
>>> a[ifancy, :][:, islice] = 100
>>> a
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
```

Is a[0,0] also 100 in this case?

Broadcasting

- NumPy operations are usually done on pairs of arrays on an element-by-element basis.
- In the simplest case, the two arrays must have exactly the same shape, as in the following example:

```
>>> a = np.array([1.0, 2.0, 3.0])
>>> b = np.array([2.0, 2.0, 2.0])
>>> a * b
array([ 2.,  4.,  6.])
```

- NumPy's broadcasting rule relaxes this constraint when the arrays' shapes meet certain constraints.
- The simplest broadcasting example occurs when an array and a scalar value are combined in an operation:

```
>>> a = np.array([1.0, 2.0, 3.0])
>>> b = 2.0
>>> a * b
array([ 2.,  4.,  6.])
```

General Broadcasting Rules

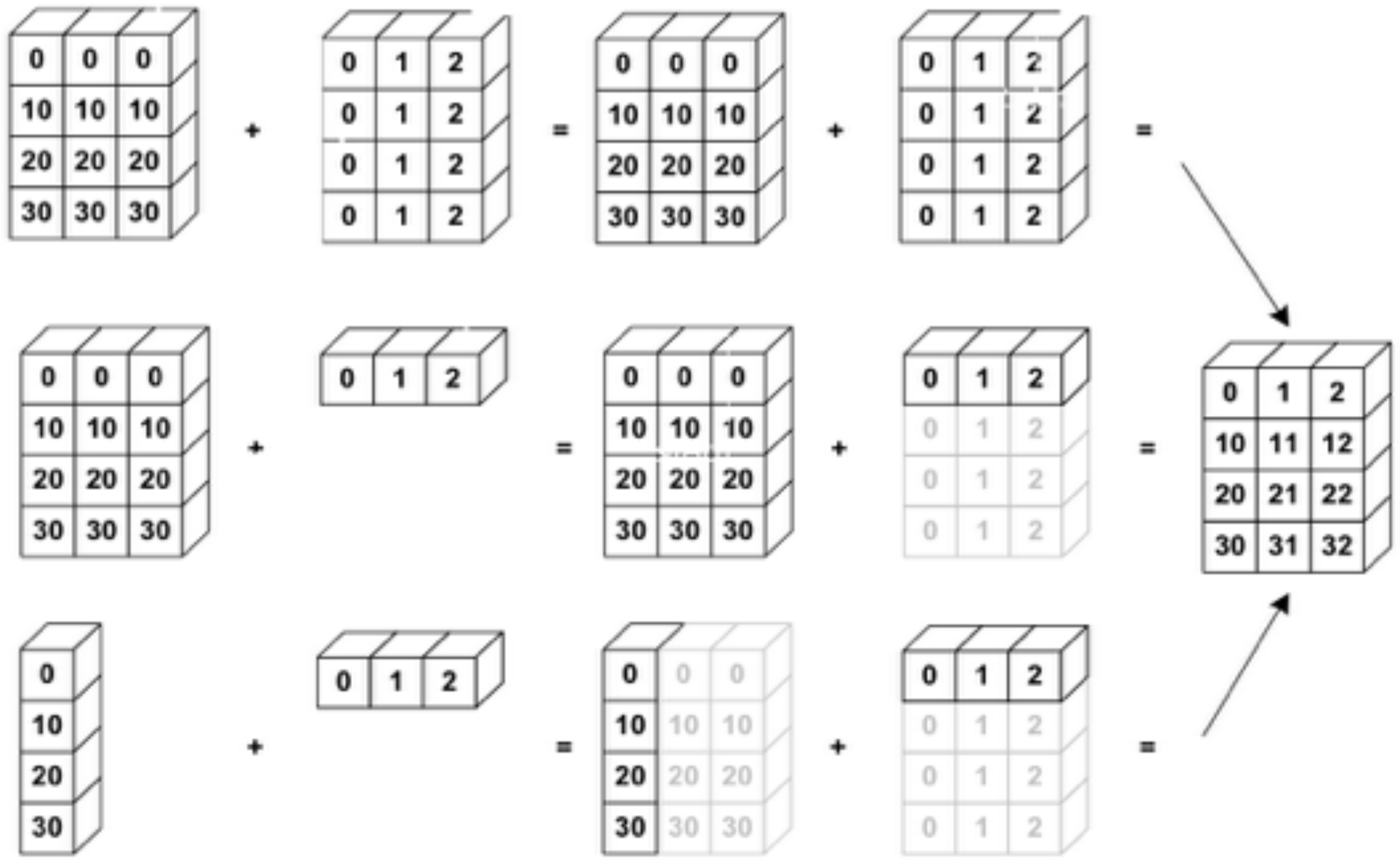
- When operating on two arrays, NumPy compares their shapes element-wise.
- It starts with the trailing dimensions, and works its way forward.
- Two dimensions are compatible when:
  1. they are equal, or
  2. one of them is 1
- The size of the resulting array is the maximum size along each dimension of the input arrays.
- Arrays do not need to have the same *number* of dimensions. For example:

```
Image  (3d array): 256 x 256 x 3
Scale  (1d array):             3
Result (3d array): 256 x 256 x 3
```

Broadcasting

- Broadcasting provides a convenient way of taking the outer product (or any other outer operation) of two arrays.
- The following example shows an outer addition operation of two 1-d arrays:

```
>>> a = np.array([0.0, 10.0, 20.0, 30.0])
>>> b = np.array([1.0, 2.0, 3.0])
>>> a[:, np.newaxis] + b
array([[  1.,   2.,   3.],
       [ 11.,  12.,  13.],
       [ 21.,  22.,  23.],
       [ 31.,  32.,  33.]])
```



Broadcasting

- Here’s a comparison between broadcasting and explicit tiling for the outer product:

```
n = 1000
a = np.arange(n)
ac = a[:, np.newaxis]
ar = a[np.newaxis, :]
%timeit np.tile(ac, (1, n)) * np.tile(ar, (n, 1))
100 loops, best of 3: 10 ms per loop
```

- And now using broadcasting

```
%timeit ar * ac
100 loops, best of 3: 2.36 ms per loop
```

- It’s also much shorter to write!

Broadcasting Exercises

```
A   A   (4d array):  8 x 1 x 6 x 1
    B   (3d array):      7 x 1 x 5

B   A   (2d array):  5 x 4
    B   (1d array):      1

C   A   (1d array):  3
    B   (1d array):  4

D   A   (2d array):      2 x 1
    B   (3d array):  8 x 4 x 3

E   A   (3d array):  15 x 3 x 5
    B   (3d array):  15 x 1 x 5
```

socrative
UU1

Which of these arrays pairs broadcasts correctly?

Boolean Operations

```
>>> import numpy as np

>>> a = np.array([1, 1, 0, 0], dtype=bool)
>>> b = np.array([1, 0, 1, 0], dtype=bool)

>>> np.logical_or(a, b)
array([ True,  True,  True, False], dtype=bool)
>>> a + b
array([ True,  True,  True, False], dtype=bool)
>>> a | b
array([ True,  True,  True, False], dtype=bool)

>>> np.logical_and(a, b)
array([ True, False, False, False], dtype=bool)
>>> a * b
array([ True, False, False, False], dtype=bool)
>>> a & b
array([ True, False, False, False], dtype=bool)

>>> np.logical_not(a)
array([False, False,  True,  True], dtype=bool)
>>> -a
array([False, False,  True,  True], dtype=bool)
>>> ~a
array([False, False,  True,  True], dtype=bool)
```

There are many ways to express boolean operation in NumPy.

Avoid Unnecessary Array Copies

The following function can be used to check if two arrays share the same data:

```
def id(x):
    # This function returns the memory
    # block address of an array.
    return x.__array_interface__[‘data'][0]

a = np.zeros(10);
aid = id(a);

b = a.copy();
id(b) == aid
>>> False
```

Avoid Unnecessary Array Copies

Array computations can involve in-place operations (the array is modified):

```
a *= 2; id(a) == aid
>>> True
```

or implicit-copy operations (a new array is created):

```
c = a * 2; id(c) == aid
>>> False
```

Be sure to choose the type of operation you actually need.

Implicit-copy operations are significantly slower, as shown here:

```
%%timeit a = np.zeros(10000000)
a *= 2
>>> 10 loops, best of 3: 19.2 ms per loop

%%timeit a = np.zeros(10000000)
b = a * 2
>>> 10 loops, best of 3: 42.6 ms per loop
```

How It Works?

- A NumPy array is basically described by metadata (number of dimensions, shape, data type, and so on) and the actual data.
- The data is stored in a homogeneous and contiguous block of memory, at a particular address in system memory (Random Access Memory, or RAM).
- This block of memory is called the data buffer. This is the main difference with a pure Python structure, like a list, where the items are scattered across the system memory.
- This aspect is the critical feature that makes NumPy arrays so efficient.

How It Works?

Why is this so important? Here are the main reasons:

1. Array computations can be written very efficiently in a low-level language like C (and a large part of NumPy is actually written in C). Knowing the address of the memory block and the data type, it is just simple arithmetic to loop over all items, for example. There would be a significant overhead to do that in Python with a list.

2. Spatial locality in memory access patterns results in significant performance gains, notably thanks to the CPU cache. Indeed, the cache loads bytes in chunks from RAM to the CPU registers. Adjacent items are then loaded very efficiently (sequential locality, or locality of reference).

3. Data elements are stored contiguously in memory, so that NumPy can take advantage of vectorized instructions on modern CPUs, like Intel's SSE and AVX, AMD's XOP, and so on. For example, multiple consecutive floating point numbers can be loaded in 128, 256 or 512 bits registers for vectorized arithmetical computations implemented as CPU instructions.

What Is The Difference Between In-Place And Implicit-Copy Operations?

- An expression like `a *= 2` corresponds to an in-place operation, where all values of the array are multiplied by two.
- By contrast, `a = a * 2` means that a new array containing the values of `a * 2` is created, and the variable a now points to this new array.
- The old array becomes unreferenced and will be deleted by the garbage collector.
- No memory allocation happens in the first case, contrary to the second case.
- More generally, expressions like `a[i:j]` are views to parts of an array: they point to the memory buffer containing the data.
- Modifying them with in-place operations changes the original array. Hence, `a[:] = a * 2` results in an in-place operation, unlike `a = a * 2`.

What Is The Difference Between In-Place And Implicit-Copy Operations?

Knowing this subtlety of NumPy can help you fix some bugs (where an array is implicitly and unintentionally modified because of an operation on a view), and optimize the speed and memory consumption of your code by reducing the number of unnecessary copies.

```
import numpy as np

a = np.array([1])
b = np.array([0])

print(a[0]+1.5)

b = a[:]
a += 1.5

print(a[0])
print(b[0])
```

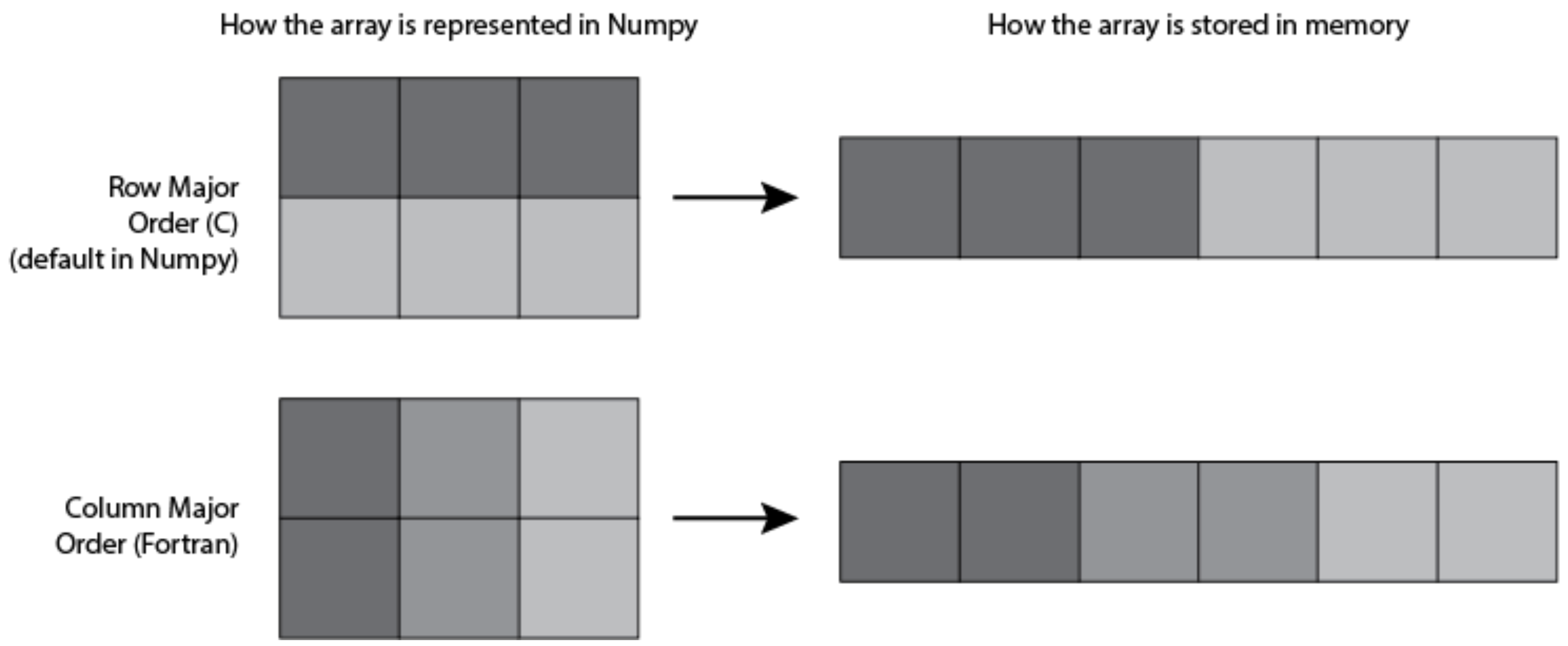
socrative
UU1

What are the results of the print statements?

Why Some Arrays Cannot Be Reshaped Without A Copy?

- A 2D matrix contains items indexed by two numbers (row and column), but it is stored internally as a 1D contiguous block of memory, accessible with a single number.
- There is more than one way of storing matrix items in a 1D block of memory: we can put the elements of the first row first, the second row then, and so on, or the elements of the first column first, the second column then, and so on.
- The first method is called row-major order, whereas the latter is called column-major order.
- Choosing between the two methods is only a matter of internal convention: NumPy uses the row-major order, like C, but unlike FORTRAN.

Why Some Arrays Cannot Be Reshaped Without A Copy?


How the array is represented in Numpy
How the array is stored in memory
Row Major Order (C) (default in Numpy)
Column Major Order (Fortran)

Why Some Arrays Cannot Be Reshaped Without A Copy?

- More generally, NumPy uses the notion of strides to convert between a multidimensional index and the memory location of the underlying (1D) sequence of elements.
- The specific mapping between array[i1, i2] and the relevant byte address of the internal data is given by

```
offset = array.strides[0] * i1 + array.strides[1] * i2
```

- When reshaping an array, NumPy avoids copies when possible by modifying the strides attribute.
- For example, when transposing a matrix, the order of strides is reversed, but the underlying data remains identical.
- However, flattening a transposed array cannot be accomplished simply by modifying strides (try it!), so a copy is needed.

Why Some Arrays Cannot Be Reshaped Without A Copy?

- Internal array layout can also explain some unexpected performance discrepancies between very similar NumPy operations.
- As a small exercise, can you explain the following benchmarks?

```
a = np.random.rand(5000, 5000)
%timeit a[0,:].sum()
%timeit a[:,0].sum()
100000 loops, best of 3: 9.57 µs per loop
10000 loops, best of 3: 68.3 µs per loop
```

Basics Of Cython

- The fundamental nature of Cython can be summed up as follows: Cython is Python with C data types.
- Almost any piece of Python code is also valid Cython code. The Cython compiler will convert it into C code which makes equivalent calls to the Python/C API.
- But Cython is much more than that, because parameters and variables can be declared to have C data types.
- Code which manipulates Python values and C values can be freely intermixed, with conversions occurring automatically wherever possible.
- Reference count maintenance and error checking of Python operations is also automatic, and the full power of Python's exception handling facilities, including the try-except and try-finally statements, is available to you – even in the midst of manipulating C data.

Cython Hello World

- As Cython can accept almost any valid python source file, one of the hardest things in getting started is just figuring out how to compile your extension.
- So lets start with the canonical python hello world. We'll save it under **`helloworld.pyx`**:

```
print(“Hello World”)
```

- Now we need to create a **`setup.py`** to compile this:

```
from distutils.core import setup
from Cython.Build import cythonize

setup(
    ext_modules = cythonize("helloworld.pyx")
)
```

- To now build your Cython file do:

```
$ python setup.py build_ext --inplace
```

- Which will leave a file in your local directory called helloworld.so in unix or helloworld.pyd in Windows.

Cython Hello World

- Now to use this file: start the python interpreter and simply import it as if it was a regular python module:

```
>>> import helloworld
Hello World
```

- Congratulations! You now know how to build a Cython extension.

- But so far this example doesn't really give a feeling why one would ever want to use Cython, so lets create a more realistic example.

Cython Primes

primes.py

```python
import numpy

"""
 Calculates first kmax primes
"""
def primes(kmax):
    p = numpy.zeros((1000),dtype=numpy.int)
    result = []
    if kmax > 1000:
        kmax = 1000
    k = 0
    n = 2
    while k < kmax:
        i = 0
        while i < k and n % p[i] != 0:
            i = i + 1
        if i == k:
            p[k] = n
            k = k + 1
            result.append(n)
        n = n + 1
    return result
```

cy_primes.pyx

```cython
def primes(int kmax):
    cdef int n, k, i
    cdef int p[1000]
    result = []
    if kmax > 1000:
        kmax = 1000
    k = 0
    n = 2
    while k < kmax:
        i = 0
        while i < k and n % p[i] != 0:
            i = i + 1
        if i == k:
            p[k] = n
            k = k + 1
            result.append(n)
        n = n + 1
    return result
```

Cython Primes

```
In [1]: import primes

In [2]: import cy_primes

In [3]: %timeit primes.primes(2000)
1 loop, best of 3: 704 ms per loop

In [4]: %timeit cy_primes.primes(2000)
100 loops, best of 3: 1.92 ms per loop
```

For certain code Cython give massive performance gains!