Topics in Computational Economics

Lecture 5

John Stachurski

NYU 2016

Comments

HW Set 2: See what other people have done at

http://nbviewer.jupyter.org/github/jstac/quantecon_nyu_2016_homework/tree/master/hw_set_2/

Please make better use of Jupyter notebooks

If you submitted, make sure your file is there!

Also, please don't use Python 2.x!

Other work: See the issue tracker for discussion of

- class presentations
- class projects

Today's Lecture

- Overview
- NumPy
- Matplotlib
- SciPy
- Cython and Numba

Overview of Scientific Computing

The scientific libraries save us time in two ways

First, there's code reuse

We don't have to write our own routines to

- sum an array
- plot a histogram
- etc., etc.

Second, the scientific libraries give us speed

- Fast execution from within a high productivity Python environment

Speed vs Productivity

Python is optimized for humans

Programmers can leave many details to the runtime environment

- specifying variable types
- memory allocation/deallocation, etc.

Hence faster to write, less error prone, easier to read

Downside: Python is harder to optimize than C / Fortran

The standard implementation of Python (called CPython) cannot match the speed of compiled languages such as C or Fortran

Does that mean that we should just switch to C or Fortran for everything?

Answer: no, no and one thousand times no

Reason:

1. For most coding, Python gives us higher productivity
2. Of any given program, very few lines are actually time-critical
3. These few lines can be tweaked to achieve C-like speeds

Details follow

Background

What are the bottlenecks to speed?

Consider an operation like

```
In [2]: a + b
```

The interpreter has to know which operation to invoke

- if strings or lists, then concatenate
- if floats, then add, etc.

Hence Python must check type and then call the correct operation

This involves substantial overheads

As we know, compiled languages avoid these overheads with explicit, static types

```c
int sum_ints(int n) {
    int i;
    int sum = 0;
    for (i = 1; i <= n; i++) {
        sum = sum + i;
    }
    return sum;
}
```

Explicit types means meaning of addition is unambiguous

Another drag on speed for high level languages is data access / memory traffic

Example. An array of ints in C is stored in a single contiguous block of memory

Successive data points can be accessed by shifting forward in memory space by a fixed amount

Python tries to replicate these ideas to some degree

- List elements are placed in contiguous memory locations

But list elements are pointers to data, not actual data

Overhead involved in accessing the data values themselves

How do we use the scientific libraries to get faster execution?

The two most important techniques are

1. Vectorization
2. Faster loops through static or JIT compiled code

(Parallelization is also important but set aside for now)

Vectorization means sending batches of related operations to native machine code

- machine code typically compiled from C or Fortran
- batch operations are usually operations on arrays

In Python, vectorized array processing usually done with NumPy

NumPy

NumPy defines an array data type called a `numpy.ndarray`

Powers a large proportion of the scientific Python ecosystem

```
In [1]: import numpy as np

In [2]: a = np.zeros(3)   # Create ndarray of 3 zeros

In [3]: a
Out[3]: array([ 0.,  0.,  0.])

In [4]: type(a)
Out[4]: numpy.ndarray
```

NumPy arrays are like native Python lists, except that

- Data must be homogeneous
- Must be one of the data types (dtypes) provided by NumPy

Example.

- float64: 64 bit floating point number
- int64: 64 bit integer

On modern machines, the default dtype is `float64`

```
In [7]: a = np.zeros(3)
In [8]: type(a[0])
Out[8]: numpy.float64
```

Creating arrays:

```
In [8]: a = np.empty(10)  # initialize empty array

In [9]: a = np.linspace(0, 2, 10)  # regular grid on [0, 2

In [10]: a = np.random.randn(4)  # 4 standard normals

In [11]: a = np.array([2, 4, 6])  # array from list

In [12]: a = np.ones(5)  # array of 5 ones
```

These are **flat** arrays with no dimension

```
In [13]: a.shape
Out[13]: (5,)
```

Let's create some multidimensional arrays

```
In [17]: A = np.empty((2, 2))

In [18]: A = np.identity(2)
In [19]: A
Out[19]:
array([[ 1.,  0.],
       [ 0.,  1.]])

In [20]: A = np.array([[2, 4],
   ....:               [6, 8]])
In [21]: A
Out[21]:
array([[2, 4],
       [6, 8]])
```

We can also adjust dimensions ex post

```
n [22]: a = np.random.randn(4)

In [23]: a
Out[23]: array([ 1.37142079, -0.76968099, -0.06442452,  1.

In [24]: a.shape
Out[24]: (4,)

In [25]: a.shape = 2, 2

In [26]: a
Out[26]:
array([[ 1.37142079, -0.76968099],
       [-0.06442452,  1.3205145 ]])
```

Array Methods:

```
In [53]: A = np.array((4, 3, 2, 1))

In [57]: A.sum()                    # Sum
Out[57]: 10

In [58]: A.mean()                   # Mean
Out[58]: 2.5

In [59]: A.max()                    # Max
Out[59]: 4
```

See also: `A.sort()`, `A.var()`, `A.cumsum()`, `A.transpose()`, `A.argmax()`, `A.std()`, etc.

How fast are these routines relative to pure Python?

See `numpy_timing.ipynb` in current lecture directory

Array Access and Slices

```
In [30]: z = np.linspace(1, 2, 5)

In [31]: z
Out[31]: array([ 1.  ,  1.25,  1.5 ,  1.75,  2.  ])

In [32]: z[0]
Out[32]: 1.0

In [33]: z[0:2]  # Two elements, starting at 0
Out[33]: array([ 1.  ,  1.25])

In [34]: z[-1]  # Last element
Out[34]: 2.0
```

Two dimensions:

```
In [35]: z = np.array([[1, 2], [3, 4]])

In [36]: z
Out[36]:
array([[1, 2],
       [3, 4]])

In [37]: z[0, 0]
Out[37]: 1

In [38]: z[0, 1]
Out[38]: 2

In [39]: z[0,:]  # All of row 0, as a flat array
Out[39]: array([1, 2])
```

Algebraic Operations

Algebraic operators act elementwise on arrays:

```
In [75]: a = np.array([1, 2, 3, 4])

In [76]: b = np.array([5, 6, 7, 8])

In [77]: a + b  # Add elementwise
Out[77]: array([ 6,  8, 10, 12])

In [78]: a * b  # Multiply elementwise
Out[78]: array([ 5, 12, 21, 32])

In [82]: a * 10  # Scalar multiplication
Out[82]: array([10, 20, 30, 40])
```

Matrix multiplication:

```
In [5]: A = np.array((1, 2))

In [6]: B = np.array((10, 20))

In [7]: A @ B
Out[7]: 50
```

Older versions of Python / NumPy use `np.dot`

```
In [7]: np.dot(A, B)
Out[7]: 50
```

Comparisons and Conditions:

```
In [97]: z = np.array([2, 3])

In [98]: y = np.array([2, 3])

In [99]: z == y
Out[99]: array([ True,  True], dtype=bool)

In [100]: y[0] = 5

In [101]: z == y
Out[101]: array([False,  True], dtype=bool)

In [102]: z != y
Out[102]: array([ True, False], dtype=bool)
```

Comparison against scalars:

```
In [8]: z = np.array((3, 6, 9))

In [9]: z == 3
Out[9]: array([ True, False, False], dtype=bool)

In [10]: z < 8
Out[10]: array([ True,  True, False], dtype=bool)
```

Conditional extraction:

```
In [11]: z[z < 8]
Out[11]: array([3, 6])
```

NumPy provides versions of many $\mathbb{R} \to \mathbb{R}$ functions that act **elementwise** on arrays

```
In [1]: z = np.array([1, 2, 3])

In [2]: np.sin(z)
Out[2]: array([ 0.84147098,  0.90929743,  0.14112001])

In [3]: (1 / np.sqrt(2 * np.pi)) * np.exp(- 0.5 * z**2)
Out[3]: array([ 0.24197072,  0.05399097,  0.00443185])
```

In NumPy,

- such functions called **universal functions**
- or just **ufuncs**

NumPy arrays are mutable

```
In [21]: a = np.array([42, 44])

In [22]: a
Out[22]: array([42, 44])

In [23]: a[-1] = 0  # Change last element to 0

In [24]: a
Out[24]: array([42,  0])

In [25]: a[:] = 1

In [26]: a
Out[26]: array([1, 1])
```

We can alter data using any reference

```
In [16]: a = np.random.randn(3)

In [17]: a
Out[17]: array([ 0.69695818, -0.05165053, -1.12617761])

In [18]: b = a

In [19]: b[0] = 0.0

In [20]: a
Out[20]: array([ 0.        , -0.05165053, -1.12617761])
```

What if you want to make an independent (i.e., **deep**) copy?

```
In [15]: a
Out[15]: array([ 0.67357176, -0.16532174,  0.36539759])

In [16]: b = np.empty_like(a)  # empty, same shape as a

In [17]: np.copyto(b, a)  # copy to b from a

In [18]: b
Out[18]: array([ 0.67357176, -0.16532174,  0.36539759])

In [19]: b[:] = 1  # Change all elements of b to 1

In [20]: a          # a is not changed
Out[20]: array([ 0.67357176, -0.16532174,  0.36539759])
```

NumPy includes some **subpackages** that provide extra functionality

For example, `np.random` used to generate RVs

```
In [134]: z = np.random.randn(10000)
```

Another subpackage is `np.linalg`

```
In [131]: A = np.array([[1, 2], [3, 4]])

In [132]: np.linalg.det(A)
Out[132]: -2.0000000000000004
```

Note however that `scipy.linalg` is more complete

Plotting

For my money, Matplotlib still the most useful for research

- Flexible and customizable
- 2D and 3D, many plot types
- LaTeX integration
- High quality output in PDF, PNG, etc.

See `http://quant-econ.net/py/matplotlib.html`

Other options to consider

- `https://plot.ly/`
- `http://pbpython.com/visualization-tools-1.html`

SciPy

A library of scientific computing routines built on top of NumPy

In fact SciPy imports everything from NumPy into its namespace

```
# inside SciPy's initialization file:
from numpy import *
from numpy.random import rand, randn
```

But few people use this — better to import NumPy explicitly

The usefulness of SciPy is in its subpackages

```
from scipy.integrate import quad
```

Subpackages:

- `scipy.optimize`
- `scipy.integrate`
- `scipy.interpolate`
- `scipy.linalg`
- `scipy.stats`
- `scipy.sparse`
- `scipy.sparse.linalg`
- `scipy.misc`
- `scipy.special`
- etc.

For an introduction see

http://quant-econ.net/py/scipy.html

For documentation see

http://docs.scipy.org/doc/scipy/reference/

Cython

Cython is

- A language specification that looks like Python with static types
  - A superset of the Python language
  - C-style type declarations
- An implementation that
  - converts Cython code into C code
  - compiles it
  - provides glue code allowing the compiled code to be called from Python

An artificial example:

Suppose that we want to compute the sum $\sum_{i=0}^{n-1} \alpha^i$ for given α, n

Suppose further that we've forgotten the basic formula

$$\sum_{i=0}^{n-1} \alpha^i = \frac{1-\alpha^n}{1-\alpha}$$

Hence we've resolved to rely on a loop

Let's look at

- a Python version
- a C version
- a Cython version

A pure Python version

```
def geo_prog(alpha, n):
    current = 1.0
    sum = current
    for i in range(n):
        current = current * alpha
        sum = sum + current
    return sum
```

A C version

```
double geo_prog(double alpha, int n) {
    double current = 1.0;
    double sum = current;
    int i;
    for (i = 1; i <= n; i++) {
        current = current * alpha;
        sum = sum + current;
    }
    return sum;
}
```

A Cython version

```
def geo_prog_cython(double alpha, int n):
    cdef double current = 1.0
    cdef double sum = current
    cdef int i
    for i in range(n):
        current = current * alpha
        sum = sum + current
    return sum
```

For details on how to compile see
http://quant-econ.net/py/need_for_speed.html

Numba

Cython produces statically compiled extensions prior to execution

Compilation is relatively straightforward because types are static and declared (just like C)

Numba uses an alternative approach called **just-in-time** compilation

Compilation occurs at run-time, rather than ahead of time

Moreover, Numba works on Python itself (unlike Cython)

In particular, no type declarations are required

Instead, types are **inferred** from function calls

See `http://quant-econ.net/py/need_for_speed.html` for more discussion