Tutorial: Numerical Python

Konrad Hinsen
Centre de Biophysique Moléculaire (CNRS)
Orléans, France

7 July 2001

Numerical Python

Ingredients:

- array type with arithmetic and comparison
- “universal” math function type
- structural operations on arrays
- linear algebra routines (using LAPACK)
- random array generator (using RANLIB)

Applications:

- matrix computations, data analysis (replacing Matlab etc.)
- building block for higher-level scientific classes (e.g. Scientific Python)
- interface to extension modules in C or Fortran

Array operations are efficient!

Arrays

Properties:

- multidimensional
- all elements of same type
- compact storage
- define arithmetic operations
- flexible indexing options
- indexing creates references

Attributes:

- **type:** character/byte, short/long integer, short/long float, short/long complex, Python object (not discussed here)
- **shape:** number of items along each axis

Conceptual model:
list [of lists [of lists ...]]

Creating arrays

```
from Numeric import *

# initialized to zero
print zeros((2, 3), Int)
```

```
[[0 0 0]
 [0 0 0]]
```

```
# from nested lists
print array([[1, 2], [-1, -2]])
```

```
[[ 1  2]
 [-1 -2]]
```

```
# as a range
print arange(0., 2., 0.4)
```

```
[ 0.   0.4  0.8  1.2  1.6]
```

Creating arrays

```
# as a function of indices
def element(i, j):
    return i-j
print fromfunction(element, (4, 3))
```

```
[[ 0 -1 -2]
 [ 1  0 -1]
 [ 2  1  0]
 [ 3  2  1]]
```

```
# using structural operations
print resize([0, 1], (3,3))
```

```
[[0 1 0]
 [1 0 1]
 [0 1 0]]
```

Array operations

```
import Numeric

a = Numeric.array([[1, 2],
                   [3, 4],
                   [5, 6]])
print a
```

```
[[1 2]
 [3 4]
 [5 6]]
```

```
print a.typecode()
```

```
l
```

```
print a.shape
```

```
(3, 2)
```

```
# extract row
print a[0]
```

```
[1 2]
```

Array operations

```
# loop over rows
for row in a:
    print row
```

```
[1 2]
[3 4]
[5 6]
```

```
# extract element
print a[1, 1]
```

```
4
```

```
# extract column
print a[:, 0]
```

```
[1 3 5]
```

```
# shape of column
print a[:, 0].shape
```

```
(3,)
```

Slices

```
from Numeric import *
a = arange(10)

# indices outside bounds
print a[6:25]
```

```
[6 7 8 9]
```

```
# every second element
print a[::2]
```

```
[0 2 4 6 8]
```

```
# reverse
print a[::-1]
```

```
[9 8 7 6 5 4 3 2 1 0]
```

```
# negative indices
print a[-2:-8:-2]
```

```
[8 6 4]
```

Complex indexing

```
from Numeric import *

a = array([[[1, 2], [3, 4]],
           [[11, 22], [33, 44]]])

# last axis
print a[..., 1]
```

```
[[ 2  4]
 [22 44]]
```

```
# first and last axis
print a[0, ..., 1]
```

```
[2 4]
```

```
# insert axis
print a[0, ..., 1][NewAxis, :]
```

```
[[2 4]]
```

```
print a[0, ..., 1][:, NewAxis]
```

```
[[2]
 [4]]
```

Array arithmetic

```
b = array([[6, 5],
           [4, 3],
           [2, 1]])

# add arrays elementwise
print a+b
```

```
[[7 7]
 [7 7]
 [7 7]]
```

```
# add row to array
print a+b[0]
```

```
[[ 7  7]
 [ 9  9]
 [11 11]]
```

```
# add number to array
print a+b[0, 0]
```

```
[[ 7  8]
 [ 9 10]
 [11 12]]
```

Array arithmetic

```
# add column to array
print a+b[:, 0, NewAxis]
```

```
[[7 8]
 [7 8]
 [7 8]]
```

```
# comparison
print a > b
```

```
[[0 0]
 [0 1]
 [1 1]]
```

```
# in-place addition
a += 3
print a
```

```
[[4 5]
 [6 7]
 [8 9]]
```

Math functions

- Module `math`: float objects
- Module `cmath`: complex objects
- Module `Numeric`: many object types
  - Numbers: int (cast to float), float, complex
  - Sequences of numbers: cast to arrays
  - Class instances: method call

Math functions

```
from Numeric import *

print sqrt(2.)
1.41421356237

print sqrt(2.+0j)
(1.41421356237+0j)

print sqrt(array([1., 2., 3.]))
[ 1.          1.41421356  1.73205081]

class Foo:
    def __init__(self, value):
        self.value = value
    def __repr__(self):
        return "Foo(%s)" % \
               repr(self.value)
    def sqrt(self):
        return Foo(sqrt(self.value))

print sqrt(Foo(2.))
Foo(1.4142135623730951)
```

Binary functions

```
from Numeric import *

a = arange(5)
b = a[::-1]
c = zeros(a.shape, a.typecode())

# standard addition
print add(a, b)
```

```
[4 4 4 4 4]
```

```
# sum over elements
print add.reduce(a)
```

```
10
```

```
# cumulative sum
print add.accumulate(a)
```

```
[ 0  1  3  6 10]
```

Binary functions

```
# outer product sum
print add.outer(a, b)

[[4 3 2 1 0]
 [5 4 3 2 1]
 [6 5 4 3 2]
 [7 6 5 4 3]
 [8 7 6 5 4]]

# addition to preallocated array
add(a, b, c)
print c

[4 4 4 4 4]

# watch out!
add(a, b, a)
print a

[4 4 4 7 8]
```

Structural operations

```
from Numeric import *

a = (1+arange(4))**2
print a
```

```
[ 1  4  9 16]
```

```
# selection by index
print take(a, [2, 2, 0, 1])
```

```
[9 9 1 4]
```

```
# selection by value
print where(a >= 2, a, -1)
```

```
[-1  4  9 16]
```

Structural operations

```
# reshaping/resizing
print reshape(a, (2, 2))
```

```
[[ 1  4]
 [ 9 16]]
```

```
print resize(a, (3, 5))
```

```
[[ 1  4  9 16  1]
 [ 4  9 16  1  4]
 [ 9 16  1  4  9]]
```

```
# element replication
print repeat(a, [2, 0, 2, 1])
```

```
[ 1  1  9  9 16]
```

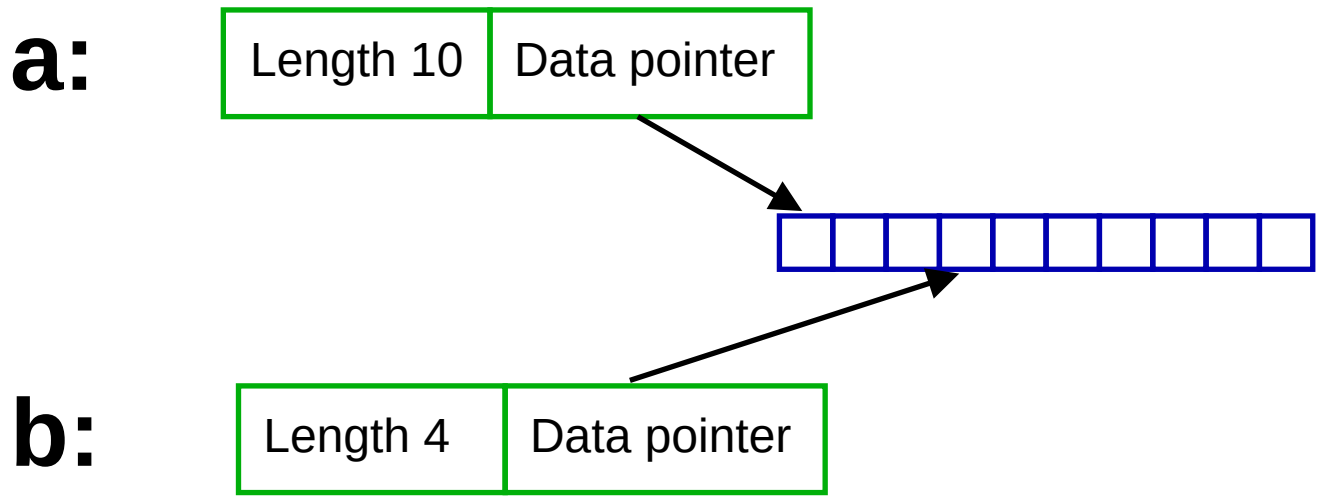
References

Indexing produces **references**, not **copies**!

```
from Numeric import *
a = arange(10)
b = a[3:7]
b[2] = 42
print a
```

```
[ 0  1  2  3  4 42  6  7  8  9]
```

Data layout in memory:



Efficiency

Individual operations:

- close to pure C code for large arrays
- significant overhead for small arrays (interpreter, array bookkeeping)

Complex calculations:

- intermediate results require allocation of temporary arrays
- time and memory overhead can be significant for very large arrays
- use in-place operations as much as possible
- move core routines to a C extension module

Interfacing to C code

Simple C function to add elements in an array:

```
double add_array(double *array, int n)
{
  double sum = 0.;
  int i;
  for (i = 0; i < n; i++)
    sum += array[i];
  return sum;
}
```

Python interface:

```
static PyObject *
addArray(PyObject *self, PyObject *args)
{
  PyArrayObject *array;
  double sum;
  if (!PyArg_ParseTuple(args, "O!",
                        &PyArray_Type, &array))
    return NULL;
  /* Type and dimension should be checked! */
  sum = add_array((double *)array->data,
                  array->dimensions[0]);
  return Py_BuildValue("d", sum);
}
```

Interfacing to Fortran code

Fortran routine:

```
      real function add(array, n)
      integer n
      real array(n)
      integer i
      real sum
      sum = 0.
      do 100 i = 1, n
         sum = sum + array(i)
100   continue
      add = sum
      end
```

Pyfort interface definition:

```
real function add (array, n)
integer n = size(array)
real array (n)
end
```

Add-on modules

Many scientific modules make use of Numerical Python:

- `LinearAlgebra` (uses LAPACK and BLAS)
- `FFT` (interface to FFTPACK)
- `fftw` (interface to FFTW)
- `RNG` (random number generator)
- ScientificPython
- PyLab
- SciPy
- ...