Numerical Python

Paul F. Dubois[1], Konrad Hinsen[2], and James Hugunin[3]

Python

Python is a small and easy to learn language with some very surprising capabilities. It is an interpreted object-oriented scripting language with a full range of sophisticated features such as first-class functions, garbage collection, and exception handling. Python has a set of properties that make it especially appealing for scientific programming:

- The language is quite simple with a very short learning curve, but it is a full and complete language.
- Extending Python with your own compiled objects and functions is very simple.
- Python is very portable, from Unix to Windows 95 to Linux to MacIntosh.
- Python is very free, with no license required even if you make a commercial product out of it.
- Python has a large user-contributed library of “modules” which cover a very wide variety of needs, such as audio and image processing, Web programming, and GUI interfaces. In particular there is an interface to the popular Tk package for building windowing applications.

1. Paul F. Dubois is a mathematician at Lawrence Livermore National Laboratory.
2. Konrad Hinsen is a physicist in the Department of Chemistry, University of Montreal.
3. James Hugunin is a graduate student in the Laboratory for Computer Science, Massachusetts Institute of Technology.

- And now, Python has a high-performance array module similar to the facilities in specialized array languages such as Matlab, IDL, Basis, or Yorick. This extension also adds complex numbers to the language. Array operations in Python lead to the execution of loops in C itself, so that most of the work is done at full compiled speed.

In this section, we'll introduce you to the Python language and the new numeric extension. More extensive tutorials and benchmarks for the basic language and the numerical extension are available on the Python Web site (http://ww.python.org).

The numeric extension is still in beta-test, and may change slightly from this description. In particular, the beta-test period is needed to sort out some controversies in naming and coercion rules. But with this tutorial as a start and the latest "readme" file for the Numerical extension, you should be able to start using it. Note that you will need to add the extension to the Python source; every effort is made to keep a "minimal Python" as small as possible, as Python is being used for applications where a small size is important.

Python is extremely suited to the kind of programmable-application development I have been advocating, with a scripting language as the user-interface, and compiled code for the compute-intensive portions.

Introducing Python

Python is an interpreter. You can either enter commands directly into the interpreter, or, more commonly, create a file containing a script. On Unix, you can invoke python with the script as the first argument, or you can use the usual trick of starting the script with a comment like this:

#!/usr/local/bin/python

Then you give the script file execute permission and when you execute it the python interpreter is invoked on the script file itself. Since the above line is a comment as far as Python is concerned, it is then ignored.

It is usual to make a Python script file have a name ending in ".py". This is required if you wish to use the file as a "module", as explained later. In the examples below, we will simply show the script and leave the process of executing it unspoken.

Hello, World

Python statements can be entered interactively at the Python prompt:

```
>>>
```

Followed by the computer response, it looks like this:

```
>>> print "Hello, World"
Hello, World
```

The print command can take a comma-delimited list of items to print, and it prints them separated by spaces and adds a “carriage return” unless the command ends in a comma. Finer control of output formatting is available. When running interactively the word “print” can be omitted, as in interactive mode the results of expressions are printed.

Expressions and Assignments

Expressions and assignments for integers and real numbers work just the way you think they do, including coercion of integers in floating-point expressions. Addition of strings means a result consisting of the concatenation of the two arguments, and post-multiplication by an integer is replication, so that one can make a string of 80 blanks with

```
blanks80 = ' ' * 80
```

There are three basic aggregate data types in Python: lists, dictionaries, and tuples.

Dictionaries are so-called “associative arrays”. That is, they can be subscripted with “keys”, usually strings. So we can remember correspondences between names and values as follows:

```
>>> Atomic_number = {} # Create an empty dictionary
>>> Atomic_number [“Hydrogen”] = 1
>>> Atomic_number [“Carbon”] = 12
>>> print Atomic_number
{‘Carbon’: 12, ‘Hydrogen’: 1}
```

A *list* is an object containing a one-dimensional list of other objects; square brackets are used to enclose the list, as in:

```
first_five_integers_list = [1, 2, 3, 4, 5]
```

The *tuple* type is similar, except that round brackets are used:

```
first_five_integers_tuple = (1, 2, 3, 4, 5)
```

In many cases Python will let you drop the parentheses on a tuple, so that this statement could also be given as

```
first_five_integers_tuple = 1, 2, 3, 4, 5
```

Long lines can be continued by ending them in a backslash. However, if inside an open parenthesis or square bracket, continuation is automatic. A singleton tuple is written (x,) to distinguish it from parentheses used for operator ordering.

Tuples are mainly created in the process of making argument lists, and can be used to return multiple values from a function or assignment:

```
a, b = 1, 2                # a == 1, b == 2
a, b = b, a + b            # a == 2, b == 3
```

The subscript operator is used to access the items in a tuple or list. The indexing is zero-based:

```
>>> x = [1.0, 2.0, 3.0, 'hello world', [1, 2]]
>>> print x[0]
1.0
>>> print x[3]
hello, world
>>> print x[4]
[1,2]
>>> print x[4][1]
2
```

The slicing operator [i: j] can also be used as a subscript. This forms a new list equal to those elements whose index is at least i and less than j.

The difference between tuples and lists is that tuples are *immutable,* meaning that the contents of a tuple cannot be changed. Lists, on the other hand, can be extended, have new elements spliced in, and have elements deleted.

For example, the following loop goes through an existing list and produces another list of those elements bigger than 2:

```
x = [2, -2, 3, 4, 0, 1, 5]
y = []
for n in x:
        if n > 2:
                y.append(n)
```

Now y contains [3, 4, 5].

To be more precise about language, y is bound to an object which is of type list with value [3, 4, 5]. Python binds names to objects, rather than having variables that contain values. So that if we now execute:

```
z = y
z[2] = 8
print z                    # Prints [3, 4, 8]
print y                    # Also prints [3, 4, 8]
```

The reason that this happens is that y and z are just two names for the same object. The statement z = y established a binding between the name “z” and the object which was already bound to the name “y”.

Suppose we bind y and z to new objects?

```
y = 1.2
z = 'z now is bound to this string'
```

What happened to the list whose value was [3, 4, 8]? That object is now unreachable from our program, since no name is bound to it. Python will reclaim the space occupied by such an object. This reclamation is called “garbage collection”.

Block Structure

Sharp-eyed readers may have been surprised at the above example:

```
for n in x:
        if n > 2:
                y.append(n)
```

What is determining the scope of the loop and the scope of the conditional statement? The answer is somewhat surprising (and unsettling, at first): it is the indentation. In other words, the “for” loop extends down to the next statement that begins at that same level of indentation or less. The same goes for defining functions. Here, for example, is a function that computes the distance between two points a and b, assuming they are passed as two-item lists or tuples.

```
from umath import sqrt    # The name sqrt means the sqrt in the math module
def euclidean_distance (a, b):
        d1 = b[0] - a[0]
        d2 = b[1] - a[1]
        return sqrt (d1**2 + d2**2)
```

The first line, “from umath import sqrt”, marks our first encounter with Python’s scalable naming conventions. Most Python input is organized into modules which are imported. In importing a module, you have your choice of whether to keep that new module’s namespace separate or merge some or all of it into your own. In this case, I merged just the name sqrt into my own namespace. I could have included all of the math functions in my name space, with “from umath import *”, or I could have kept the math functions in their own name space:

```
import umath
def euclidean_distance (a, b):
        d1 = b[0] - a[0]
        d2 = b[1] - a[1]
        return umath.sqrt (d1 * d1 + d2 * d2)
```

This name space control makes Python very suitable for creating large libraries of modules without fear of unintentional name collisions.

The indentation convention is controversial. Every few months a newcomer to Python will start a discussion about what a stupid idea this is and ask how can one live without braces or begin-end pairs or the like. The interesting thing is that one does not hear this from Python veterans. Armed with an indenting editor, such as Emacs with the Python-mode extension, the language soon becomes natural and is perceived as easy-to-read and uncluttered.

The Numerical Extension

The numerical extension to Python was the result of an Internet collaboration via a mailing list. The Python community has set up a series of such “Special Interest Groups”, or SIGs, to study areas for improvement and merging of similar efforts. The “Matrix-SIG” discussions inspired James Hugunin, a graduate student in the Laboratory for Computer Science at MIT, to implement a proposal starting from previous work done by James Fulton of the United States Geological Survey in Reston, VA. Hugunin’s work has now been released for beta-testing. His work was augmented with testing and new ideas from the members of the Matrix-SIG, complex number and other parser work by Konrad Hinsen of the University of Montreal, and enhancements for array slicing by Chis Chase of The Applied Physics Laboratory, The John Hopkins University.

The work is motivated by the observation that Python lists are not a suitable vehicle for computationally-intensive work, in that they are true lists and not represented as blocks of contiguous storage. A list of 10,000 floating-point numbers is actually a list of 10,000 floating-point objects.

It is possible (and remarkably easy) to add new object types to Python both by extensions written in C and by using the object-oriented features in Python proper. The main part of the numeric extension consists of a C-language extension which defines a new “array” object and also an object “ofunc” (for “optimized function”) which enables fast element-by-element operations on the arrays. Python has a mechanism by which such extensions can choose to implement operations in the language, such as addition, subtraction, subscripting, and subscripted assignment.

Creating Arrays

In what follows we assume that the user has done “from Numeric import *” so that all the facilities can be used without qualifying their names. The basic “constructor” for arrays is the function array, which will convert any Python object that has a sequence-like behavior (such as lists and tuples) into an array of an appropriate type. Floating-point numbers in Python correspond to C’s “double” type, i.e. usually 64 bit quantities. Integer elements in Python correspond to C “long ints”. However, the array class also supports a large number of other integer, complex, and floating precisions, as well as arrays of Python objects. Here are a few examples of constructor calls:

array([1,2,3]) -> array of integers
array([1, 2.3, 4]) -> array of doubles, since one element was double
array([1, 2j, 3.]) -> array of complex, since one element complex

Naturally, in practice most arrays are not entered literally like this but are the result of a computation or processing of a data file. There are functions provided to create common arrays:

- zeros(2,3,4) makes a zero array of shape (2,3,4)
- zeros(2,3,4, Integer()) makes an integer array of the same shape. A series of such type functions, similar in spirit to Fortran 90's kind specifiers, allows specifying arrays of different types and precisions, such as Float(32).
- ones is like zeros, only sets the elements to 1.
- arange (n) produces the array of integers from zero to n - 1.
- fromFunction([2,3,4], lambda i, j, k: 100*i + 10 *j + k) makes a (2,3,4)-shaped matrix with the (i, j, k) element computed using the "anonymous" lambda function shown. You could also give the name of a function that took three arguments and returned a value.

Arrays as Objects

Arrays are in fact objects. Everything in Python is an object, in fact. Dictionaries, lists, functions, tracebacks, strings, and so on, are all objects. As such, they may have attributes (data members, or at least what appear to be data members) and member functions. The standard object-oriented "dot" notation is used to access these attributes and functions. For example, for lists we have two member functions "append" and "reverse" which modify the list.

```
>>> x=[1, 2, 3]
>>> x.append(4)
>>> x.reverse()
>>> print x
[4, 3, 2, 1]
```

Shapes

Each array has a shape, available as an attribute of the array.

```
>>> x=arange(10)
>>> print x
0 1 2 3 4 5 6 7 8 9
>>> x.shape
(10,)
```

The shape can be changed by assigning to this attribute:

```
>>> x.shape = (2,5)
>>> print x
0 1 2 3 4
5 6 7 8 9
```

This process is “smart” and will detect the problem if the new length doesn’t match the old.

Array Expressions

Array expressions are carried out by producing a result whose components are the result of doing the operation on the corresponding elements of each operand. This “elementwise” idea is extended to various cases where the shapes of the operands are not identical, and a collection of common functions that operate elementwise on arrays are provided.

Thus, if two real arrays a and b have the same shape,

```
x = (a + b) / (a - b) + sin(a)**2 + cos(b)**2
```

results in an array with the same shape as `a` and `b`, whose individual elements are calculated using the corresponding elements of `a` and `b` in the above expression.

In general, such expressions raise an exceptions if the arrays `a` and `b` do not have the same shape. There is, however, one important exception: If an axis of an array has length one, this array will be compatible with any array that matches in the other axes. For example, and array with shape (3,1,2) can be combined with an array of shape (3,5,2) or (3,100,2). In this case, the first array will be repeated along the axis of length 1 until it matches the other array. (Of course the array elements are not actually copied in memory, but the effect is the same.)

This extension process goes even one step further. If the two arrays do not have the same number of dimensions, the lower-rank array is “upgraded” to the rank of the other one by adding axes of length one in front of its shape. For example, when an array of shape (3,2) is combined with an array of shape (5,3,2), it is considered to be of shape (1,3,2) and then repeated to have shape (5,3,2).

This extension and repetition allows a compact notation for common operations. For example, a scalar can be added to an array of arbitrary rank, with the effect of being added to each individual element. Likewise, a whole array can easily be multiplied by a scalar. In this example, the array y of shape (3,) is added to the array x of shape(4,3), with the result that y has been added to each row of x to produce the result.

```
>>> x
10 11 12
13 14 15
16 17 18
19 20 21
>>> y
0 1 2
>>> x + y
10 12 14
13 15 17
16 18 20
19 21 23
```

To make full use of this powerful combination scheme, it is often useful to add axes of length one to an array in other positions than before its shape vector. How this is done will be explained in the next section on subscripting.

Array Subscripting

The array class has elaborate facilities for accessing portions of an array. Subscripts which refer to a single element produce a result that is a scalar; other subscripts produce a reference to the subarray specified.

```
>>> y=arange(12)
>>> y.shape=(4,3)
>>> print y
 0  1  2
 3  4  5
 6  7  8
 9 10 11
>>> print y[0,0]
0
>>>print y[1,0]
3
>>> print y[0,1]
1
>>>print y[0][1]
1
>>> print y[1]
3 4 5
```

The slice operator : can also be used to indicate that some or all of a certain dimension is to be chosen. In Python, i: j means all indices from i up to but not including j. An optional :k can be added to indicate a stride count.

```
>>> print y[1:3]
3 4 5
6 7 8
 >>> print y[:, 1]
 1  4  7 10
>>>  y[:, 1] = [5,6,7,8]
>>> print y
 0  5  2
 3  6  5
 6  7  8
 9  8 11
```

Special Indexing Operations

Suppose you want to create the outer product of two vectors, a and b. That is, you want to create c such that c has shape (len (b), len (a)) and c [i, j] = a [i] * b [j]. We could accomplish this somewhat clumsily as follows:

```
>>>a = array ( [1, 2, 3])
>>> b = array ( [10, 20])
>>> c = zeros (len (b), len (a), Integer ())
>>> for i in range (len (b)):
...         for j in range (len (a)):
...                 c [i, j] = b [i] * a [j]
...
>>> print c
10 20 30
20 40 60
```

This is not just clumsy, it will be noticeably slow for vectors of longer length. While the Python interpreter is quite fast, it is still much slower than carrying out the operations in compiled code. So, we seek an array syntax to accomplish this. The broadcasting rules provide the key.

```
>>> b.shape=(2,1)
>>> print b
10
20
>>> c = a * b
>>> print c
10 20 30
20 40 60
```

Note that in writing `a * b` we have made use of the automatic repetition feature described in the previous section. All that we need to make this example more convenient is a better way to achieve the reshaping of `b`. In effect, we want to add a new axis of length one to the end of the shape vector of `b`. Since adding a new axis of length one is a very common form of reshaping, Python has a convenient shorthand notation for it. In an array subscript expression, you simply put the special index `NewAxis` at the position where you want to create a new axis. Therefore

```
b [:, NewAxis]
```

will be just what we need, and the outer product can be written compactly as

```
c = a * b [:, NewAxis]
```

Another common problem is that you might want to extract all elements along the first axes, but just the first element along the last axis. Of course this is possible with what we have explained until now. If the array a has the shape (2,2,3), the answer is:

```
a [:, :, 0]
```

But what if you need this to work for an *arbitrary* array of unknown shape, for example in a general utility function? You'd have to use one empty slice operator for each axis, but to do so you must know the number of axes. The way out is the special index ..., which stands for as many empty slice operators as is required to cover all the axes in the array. (It is clear from this definition that you can use this special index only once in a subscript expression.) The above example can therefore be written as:

```
a [..., 0]
```

Array Methods

Each array object has a set of methods that can be applied to it. These include:

- x.equal(y) returns an array of 1's and 0's of the same shape as x, indicating whether or not the corresponding elements are equal. equal has brothers notEqual, greater, greaterEqual, less, and lessEqual, and cousins andLogical, orLogical, and notLogical for carrying out logical operations. (Caution, Python's normal scalar comparison operators do not work on arrays).
- x.matrixMultiply(y) returns the mathematical matrix product of x and y.
- There are a set of methods for producing arrays derived from the given array, such as transposes, complex conjugate, copies, concatenations, and so on.
- x.choose (list), x.take (list), and x.repeat (list) are available for more complicated needs in choosing portions of an array.

Reduction

The Numeric extension, like the rest of Python, regards functions as first class objects. A set of these functions has been provided which optimize certain kinds of calculations on arrays. Most of these functions, such as sin, cos, and sqrt, are unary functions which operate elementwise. However, some of the functions are binary, the two most useful being add and multiply. Thus multiply(x,y) produces the same result as x * y. What makes the concept really useful is that add and multiply are objects, and these objects have methods. The most useful of these methods is reduce.

A reduction by a binary operator applies that binary operator repeatedly along a certain dimension (by default, the first), until that dimension has been reduced to a scalar. Thus, an object with one less dimension is produced.

```
>>> z
1.00000000 2.00000000 3.00000000
4.00000000 5.00000000 6.00000000
>>> add.reduce(z)   #sum the columns
5.00000000 7.00000000 9.00000000
>>> add.reduce(z,1)  #sum the rows
6.00000000 15.00000000
```

The way to think about the optional second argument here is that you are picking the index of the dimension that is to be eliminated. In this example, z has shape (2, 3). After the (default) reduction along the dimension numbered 0, it has shape (3,). After reduction along the dimension numbered 1, it has shape (2,).

This allows us to define an easy “dot product” function:

```
def dot (a, b):
        return add.reduce (a * b)
>>> w
1.00000000 2.00000000 3.00000000
>>> v
-3.00000000 2.00000000 2.00000000
>>> dot(w,v)
7.0
```

Object-Oriented Features

Python has a powerful mechanism for declaring your own kinds of objects. If you are doing a lot of work with matrices, you might want a Matrix object that was otherwise similar to the array object, but which implemented the multiply operator as matrix multiplication rather than element-wise multiplication. Or you might want to invent your own plotting classes which contained objects such as Curves and Surfaces, with attributes in these classes such as line_thickness, color, or label, or a Matrix used for rotation.

Python's facility for classes lets you do just that, and even lets you override the meaning of operators, attribute assignment, subscripting, and so on. Here for example is a simple class representing points with two coordinates, defining addition for them:

```
import umath
class Point:
        "Points in two space -- skeletal example."
        def __init__ (self, x, y):
                "Construct a new point with coordinates x and y"
                self.x = x
                self.y = y
        def __add__ (self, other):
                "a + b"
                return Point (self.x + other.x, self.y + other.y)
        def norm (self):
                "norm () = length of vector from origin to this point"
                return umath.sqrt (self.x**2 + self.y**2)
```

You would subsequently use this class in statements like this:

```
x = Point (1.0, 2.0)
y = Point (3.0, 6.0)
print (x+y).norm ()
```

This just scratches the surface of the class facility. If you're curious about those strings that follow the first line of each class and function, those are called "doc strings", and are visible to the user at run-time in a special attribute of the function object. Plans are to develop tools to extract them automatically from Python source for creating class library documentation, as is done in Eiffel, for example.

Classes give you the power of inheritance. A Python class has been implemented from which variants of the array object can be derived, and one such derivation is a class "Matrix" for two-dimensional objects which changes the "*" operator to mean matrix multiplication rather than elementwise multiplication. The standard array object has a method matrixMultiply so that a.matrixMultiply (b) is the mathematical matrix product.

A Programming Strategy

What you've just seen is remarkable: a complete array facility added to Python as a user-written extension. The Numeric extension is a high-performance array language with remarkably sophisticated facilities. There were some changes made to the Python parser, involving imaginary constants, the ** operator for exponentiation, and multiple slicing subscripts. These changes were only needed to improve the syntactic appearance, such as being able to write `a [i, j]` rather than `a [ (i, j)]`, and mostly were on the "to-do" list for the language anyway and didn't break any existing code.

The most powerful part of Python is its growing user community and their spirit of cooperation in contributing modules into the user-contributed library. See:

http://www.python.org/python/Contributed.html

Interfaces to such things as netCDF and PDB self-describing files, the PLPLOT graphics package, the Yorick graphics subsystem, and FFT packages are under development. We recommend asking about your area of interest on the newsgroup comp.lang.python.

The numerical extension is a particular case of a tremendously successful strategy: use Python as a scripting language, adding objects and functions written in a compiled language to it. Python's portability, free licensing, basic good performance, and ease of extension, combined with its appeal to the scientific community due to its simplicity and the ease with which application-specific objects can be created, make it a key tool for the future.