Numerical Python

David Ascher
Paul F. Dubois
Konrad Hinsen
Jim Hugunin
Travis Oliphant

Lawrence Livermore National Laboratory, Livermore, CA 94566
UCRL-MA-128569

Legal Notice

This work was produced at the University of California, Lawrence Livermore National Laboratory under contract no. W-7405-ENG-48 between the U.S. Department of Energy and The Regents of the University of California for the operation of UC LLNL.

Table of Contents

1. Introduction

This chapter introduces the Numeric Python extension and outlines the rest of the document.

The Numeric Python extensions (NumPy henceforth) is a set of extensions to the Python programming language which allow Python programmers to efficiently manipulate large sets of objects organized in grid-like fashion. These sets of objects are called arrays, and they can have any number of dimensions: one dimensional arrays are similar to standard Python sequences, two-dimensional arrays are similar to matrices from linear algebra. Note that one-dimensional arrays are also *different* from any other Python sequence, and that two-dimensional matrices are also *different* from the matrices of linear algebra, in ways which we will mention later in this text.

Why are these extensions needed? The core reason is a very prosaic one, and that is that manipulating a set of a million numbers in Python with the standard data structures such as lists, tuples or classes is much too slow. Anything which we can do in NumPy we can do in standard Python – we just may not be alive to see the program finish. A more subtle reason for these extensions however is that the kinds of operations that programmers typically want to do on arrays, while sometimes very complex, can often be decomposed into a set of fairly standard operations. This decomposition has been developed similarly in many array languages. In some ways, NumPy is simply the application of this experience to the Python language – thus many of the operations described in NumPy work the way they do because experience has shown that way to be a good one, in a variety of contexts. The languages which were used to guide the development of NumPy include the infamous APL family of languages, Basis, MATLAB, FORTRAN, S and S+, and others. This heritage will be obvious to users of NumPy who already have experience with these other languages. This tutorial, however, does not assume any such background, and all that is expected of the reader is a reasonable working knowledge of the standard Python language (as of this writing the current Python release is 1.5.1).

This document is the “official” documentation for NumPy. It is both a tutorial and the most authoritative source of information about NumPy with the exception of the source code. The tutorial material will walk you through a set of manipulations of simple, small, arrays of numbers, as well as image files. This choice was made because 1) a concrete data set makes explaining the behavior of some functions much easier to motivate than simply talking about abstract operations on abstract data sets 2) every reader will at least an *intuition* as to the meaning of the data and organization of image files, and 3) the result of various manipulations can be displayed simply since the data set has a natural graphical representation. All users of NumPy, whether interested in image processing or not, are encouraged to follow the tutorial with a working NumPy installation at their side, testing the examples, and, more importantly, transferring the understanding gained by working on images to their specific domain. The best way to learn is by doing – the aim of this tutorial is to guide you along this “doing.”

Chapter 2 provides information on testing Python, NumPy, and compiling and installing NumPy if necessary.

Chapter 3 provides information on testing and installing the NumTut package, which allows easy visualization of arrays.

Chapter 4 gives a high-level overview of the components of the NumPy system as a whole.

Chapter 5 provides a detailed step-by-step introduction to the most important aspect of NumPy, the multidimensional array objects.

Chapter 6 provides information on universal functions, the mathematical functions which operate on arrays and other sequences elementwise.

Chapter 7 covers pseudo-indices.

Chapter 8 is a catalog of each of the utility functions which allow easy algorithmic processing of arrays.

Chapter 9 discusses the methods of array objects.

Chapter 10 presents the attributes of array objects.

Chapter 11 is a collection of special topics, from the organization of the codebase to the mechanisms for customizing pringint.

Chapter 12 is an tutorial on how to write a C extension which uses NumPy arrays.

Chapter 13 is a reference for the C API to NumPy objects (both PyArrayObjects and UFuncObjects).

Chapter 14 is a reference for the Fast Fourier Transform module

Chapter 15 is a reference for the Linear Algebra module

Chapter 16 is a reference for the RandomArray random number generator module

Chapter 17 is a glossary of terms

Chapter 18 is a listing of known bugs and documentation tasks left undone.

Acknowledgments

Numerical Python is the outgrowth of a long collaborative design process carried out by the Matrix SIG of the Python Software Activity (PSA). Jim Hugunin, while a graduate student at MIT, wrote most of the code and initial documentation. When Jim joined CNRI and began working on JPython, he didn't have the time to maintain Numerical Python so Paul Dubois at LLNL agreed to become the maintainer of Numerical Python. David Ascher, working as a consultant to LLNL, wrote most of this document, incorporating contributions from Konrad Hinsen and Travis Oliphant, both of whom are major contributors to Numerical Python.

Many other people have contributed to Numerical Python by making suggestions and sending in bug fixes. Numerical Python illustrates the power of the open source software concept. We hope that readers will also send in "bug fixes" for this manual. We have made this early first release of the manual in the belief that such a process is the fastest way to improve it. Please send comments about the manual to support@icf.llnl.gov or to Paul Dubois, L-264, Lawrence Livermore National Laboratory, Livermore, CA 94566, U.S.A.

2. Installing NumPy

This chapter explains how to install and test NumPy, from either the source distribution or from the binary distribution.

Before we start with the actual tutorial, we will describe the steps needed for you to be able to follow along the examples step by step. These steps including installing Python, the NumPy extensions, and some tools and sample files used in the examples of this tutorial.

Testing the Python installation

The first step is to install Python if you haven't already. Python is available from the Python website's download directory at *http://www.python.org/download*. Click on the link corresponding to your platform, and follow the instructions described there. When installed, starting Python by typing `python` at the shell or double-clicking on the Python interpreter should give a prompt such as:

```
Python 1.5.1 (#0, Apr 13 1998, 20:22:04) [MSC 32 bit (Intel)] on win32
Copyright 1991-1995 Stichting Mathematisch Centrum, Amsterdam
>>>
```

If you have problems getting this part to work, consider contacting a local support person or emailing *python-help@python.org* for help. If neither solution works, consider posting on the comp.lang.python newsgroup (details on the newsgroup/mailing list are available at *http://www.python.org/psa/MailingLists.html#clp*).

Testing the Numeric Python Extension Installation

The standard Python distribution does not come as of this writing with the Numeric Python extensions installed, but your system administrator may have installed them already. To find out if your Python interpreter has NumPy installed, type `import Numeric` at the Python prompt. You'll see one of two behaviors (throughout this document, **`bold Courier New`** font indicates user input, and `standard Courier New` font indicates output):

```
>>> import Numeric
Traceback (innermost last):
 File "<stdin>", line 1, in ?
ImportError: No module named Numeric
>>>
```

indicating that you don't have NumPy installed, or:

```
>>> import Numeric
>>>
```

indicating that you do. If you do, go on to the next step. If you don't, you have to get the NumPy extensions.

Installing NumPy

There are currently two distributions available.

On Win32

For Microsoft Windows 95, 98 and NT, a binary installer is available at *ftp://ftp-icf.llnl.gov/pub/python/NumPy.exe*. This installer is simple to use (simply double-click on the NumPy.exe file and answer each screen in turn). Running this installer will perform all the needed modifications to your Python installation so that NumPy works.

On Unix

For both Unix and other platforms, NumPy must be compiled from the source. The source distribution for NumPy is available as part of the LLNLPython distribution, which is available at *ftp://ftp-icf.llnl.gov/pub/python/LLNLPython.tgz*. This is a gzipped tarfile. It should be uncompressed using the gunzip program and untarr'ed with the tar program:

```
csh> gunzip LLNLPython.tgz
csh> tar xf LLNLPython.tar
```

Follow the instructions in the toplevel directory for compilation and installation.

The standard Python installer for the Macintosh (available at *http://www.python.org/download/download_mac.html*) also optionally installs the NumPy extensions, although these are typically not the most up-to-date. .

If you have problems getting this part to work, consider contacting a local support person or emailing *python-help@python.org* . Alternatively, you can send a description of your problem to the Matrix-SIG (a special interest group devoted to the NumPy extension – details are available at *http://www.python.org/sigs/matrix-sig/*). For a more targeted audience, the email address *support@icf.llnl.gov* provides technical support for the LLNLPython distribution, of which NumPy is a part.

Just like all Python modules and packages, the Numeric module can be invoked using either the `import Numeric` form, or the `from Numeric import ...` form. Because most of the functions we'll talk about are in the Numeric module, all As with any other Python module, the Numeric module can be imported one of two ways, using either the "import Numeric"or the "from Numeric import *" form. In this document, all of the code samples will assume that they have been preceded by a statement:

```
from Numeric import *
```

---

3. The NumTut package

This chapter leads the user through the installation and testing of the NumTut package, which should have been distributed with this document.

Testing the NumTut package

This tutorial assumes that the NumTut package has been installed. This package contains a few sample images and utility functions for displaying arrays and the like. To find out if NumTut has been installed, do:

```
>>> from NumTut import *
>>> view(greece)
```

If a picture of a greek street shows up on your screen, you're all set, and you can go to the next chapter.

Possible reasons for failure:

```
>>> import NumTut
Traceback (innermost last):
  File "<stdin>", line 1, in ?
ImportError: No module named NumTut
```

This message indicates that you do not have the NumTut package installed in your PythonPath. NumTut is distributed along with this document, so it should be available wherever you obtained this document. The main distribution site for this document (and NumTut) is: *ftp://ftp-icf.llnl.gov/pub/python*. To installation NumTut, simply untar the NumTut.tar.gz file so that it is in your PythonPath. For example, on Win32, it can be placed in the main directory of your Python installation. On Unix, it can be placed in the site-packages directory of your installation.

Win32

```
>>> import NumTut
Traceback (innermost last):
```

```
[...]
ConfigurationError: view needs Tkinter on Win32, and either threads or
the IDLE editor"
```

or:

```
ConfigurationError: view needs either threads or the IDLE editor to be
enabled.
```

On Win32 (Windows 95, 98, NT), the Tk toolkit is needed to view the images. Additionally, either the Python interpreter needs to be compiled with thread support (which is true in the standard win32 distribution) or you need to call the NumTut program from the IDLE interactive development environment.

If you do not wish to modify your Python installation to match these requirements, you can simply ignore the references to the demonstrations which use the `view()` command later in this document. Using NumPy does not require image display tools, they just make some array operations easier to understand.

Unix

On Unix machines, NumTut will work best with a Python interpreter with support (not true in the default configuration), with the Tkinter GUI framework available and optionally with the tkImaging add-on (part of the Python Imaging Library). If this is not the case, it will try to use an external viewer which is able to read PPM files. The default viewer is 'xv' a common image viewer available from *ftp://ftp.cis.upenn.edu/pub/xv*. If xv is not installed, you will get an error message similar to:

```
>>> import NumTut
Traceback (innermost last):
[...]
ConfigurationError: PPM image viewer 'xv' not found
```

You can configure NumTut to use a different image viewer, by typing e.g.:

```
>>> import NumTut
>>> NumTut.view.PPMVIEWER = 'ppmviewer'
>>> from NumTut import *
>>> view(greece)
```

If you do not have a PPM image viewer, you can simply ignore the references to the demonstrations which use the `view()` command later in this document. Using NumPy does not require image display tools, they just make some array operations easier to understand.

4. High-Level Overview

In this chapter, a high-level overview of the extensions is provided, giving the reader the definitions of the key components of the system. This section defines the concepts used by the remaining sections.

Numeric Python consists of a set of modules:

- `Numeric.py` (and its helper modules `multiarray`, `umath` and `fast_umath`)

  This module defines two new object types, and a set of functions which manipulate these objects, as well as convert between them and other Python types. The objects are the new array object (technically called `multiarray` objects), and universal functions (technically `ufunc` objects).

- `RandomArray.py` (and its helper module `ranlib`)

  This module provides a high-level interface to a random-number generator.

- `FFT.py` (and its helper module `fftpack`)

  This module provides a high-level interface to the fast Fourier transform routines implemented in the `FFTPACK` library if it is available, or to the compatible but less optimized fftpack library which is shipped with Numeric Python..

- `LinearAlgebra.py` (and its helper module `lapack_litemodule`)

  This module provides a high-level interface to the linear algebra routines implemented in the `LAPACK` library if it is available, or to the compatible but less optimized `lapack_lite` library which is shipped with Numeric Python.

Array Objects

The array objects are generally homogeneous collections of potentially large numbers of numbers. All numbers in a multiarray are the same kind (i.e. number representation, such as double-precision floating point). Array objects must be full (no empty cells are allowed), and their size is immutable. The specific numbers within them can change throughout the life of the array. The shape of an array is the set of numbers which describe the number of indices of the array in each dimension. The total number of dimensions of an array is called its rank, and is the number of numbers which make up the shape. A vector is a rank-1 array (it has only one dimension along which it can be indexed). A matrix as used in linear algebra is a rank-2 array (it has two dimensions along which it can be indexed). There are also rank-0 arrays, which can hold single scalars -- they have no dimension along which they can be indexed, but they contain a single number. Mathematical operations on arrays return new arrays containing the results of these operations performed *elementwise* on the arguments of the operation.

The *size* of an array is the total number of elements therein (it can be 0 or more). It does not change throughout the life of the array.

The *shape* of an array is a description of the number of dimensions of the array and its extent in each of these dimensions (it can be 0, 1 or more). It can change throughout the life of the array.

The *rank* of an array is the number of dimensions along which it is defined. It can change throughout the life of the array.

The *typecode* of an array is a single character description of the kind of element it contains (number format, character or Python reference). It determines the itemsize of the array.

The *itemsize* of an array is the number of 8-bit bytes used to store a single element in the array. The total memory used by an array tends to its size times its itemsize, as the size goes to infinity (there is a fixed overhead per array, as well as a fixed overhead per dimension.

Here is an example of Python code using the array objects (bold text refers to user input, non-bold text to computer output):

```
>>> from Numeric import array
>>> vector1 = array((1,2,4,5))
>>> print vector1
[1 2 3 4 5]
>>> matrix1 = array(([0,1],[1,3]))
>>> print matrix1
[[0 1]
 [1 3]]
>>> print vector1.shape, matrix1.shape
(5,) (2,2)
>>> print vector1 + vector1
[ 2  4  6  8  10]]
>>> print matrix1 * matrix1
[[0 1]                          # note that this is not the matrix
 [1 9]]                         # multiplication of linear algebra
```

Universal Functions

Universal functions (ufuncs) are functions which operate on arrays and other sequences. Most ufuncs perform mathematical operations on their arguments, also elementwise.

Here is an example of Python code using the ufunc objects:

```
>>> from Numeric import greater, add
>>> print greater([1,2,4,5], [5,4,3,2])
[0 0 1 1]
>>> print add([1,2,4,5], [5,4,3,2])
[6 6 7 7]
>>> print add.reduce([1,2,4,5])
12                              # 1 + 2 + 3 + 4 + 5
```

Ufuncs are covered in detail in “Ufuncs” on page 35.

Convenience Functions

The Numeric module provides, in addition to the functions which are needed to create the objects above, a set of powerful functions to manipulate arrays, select subsets of arrays based on the contents of other arrays, and other array-processing operations.

```
>>> from Numeric import arange, where, greater
>>> data = arange(10)           # convenient homolog of builtin
range()
>>> print data
[0 1 2 3 4 5 6 7 8 9]
>>> print where(greater(data, 5), -1, data)
[ 0  1  2  3  4  5 -1 -1 -1 -1]  # selection facility
>>> data = resize(array((0,1)), (9, 9))
>>> print data
[[0 1 0 1 0 1 0 1 0]
 [1 0 1 0 1 0 1 0 1]
```

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

All of the functions which operate on NumPy arrays are described in “Array Functions” on page 43.

RandomArray

The `RandomArray` module provides a high-level interface to the ranlib number generator. It provides a uniform distribution generator of pseudo-random numbers, as well as some convenience functions:

```
>>> from RandomArray import random, uniform, randint, permutation
>>> print random((5,5))
[[ 0.45456091  0.53438765  0.72412336  0.12156525  0.79255972]
 [ 0.14763653  0.93401444  0.38913983  0.97293309  0.45860398]
 [ 0.57528652  0.9801351   0.19893601  0.3396503   0.12224415]
 [ 0.9067847   0.37667559  0.71613152  0.24334284  0.68907028]
 [ 0.9655151   0.29746972  0.42734603  0.72314573  0.66344323]]
>>> print uniform(-1.0,1.0, (5,))
[-0.2637264   0.12331069  0.11497829 -0.25969645  0.36571342]
>>> print randint(10, 20, (4,2))
[[19 14]
 [14 11]
 [13 11]
 [13 11]]
>>> print permutation(10)
[0 5 9 4 2 1 6 8 3 7]
>>> print permutation(10)
[3 7 1 2 9 0 4 8 5 6]
```

The reader should also be aware that LLNL provides an alternative random number generator, called RNG, which also provides normal, log-normal and exponential distribution number generators (XXX other differences?). It is available at *ftp://ftp.icf.llnl.gov/pub/python/XXX*. See “RandomArray Reference” on page 86 for details.

FFT

The FFT module provides a high-level interface to the fast Fourier transform routines which are implemented in the FFTPACK library. It performs one and two-dimensional FFT’s, forward and backwards (inverse FFTs), and includes efficient routines for FFTs of real-valued arrays. It is most efficient for arrays whose size is a power of two.

```
>>> from FFT import fft, inverse_fft
>>> data = array((1.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0))
>>> print data
[ 1.  0.  0.  0.  1.  0.  0.  0.]
>>> print fft(data)
[ 2.+0.j  0.+0.j  2.+0.j  0.+0.j  2.+0.j  0.+0.j  2.+0.j  0.+0.j]
>>> print inverse_fft(fft(data))
[ 1.+0.j  0.+0.j  0.+0.j  0.+0.j  1.+0.j  0.+0.j  0.+0.j  0.+0.j]
```

See “FFT Reference” on page 81 for details.

LinearAlgebra

The LinearAlgebra module provides a high-level interface to the most commonly used functionality of the LAPACK library, in a Python-friendly fashion. It includes functions to solve systems of linear equations and linear least squares problems, invert matrices, compute eigenvalues and eigenvectors, generalized inverses, determinants, as well as perform singular value decomposition.

```
>>> from LinearAlgebra import inverse
>>> data = array(((1.0,2), (4,5)))
>>> print data
[[ 1.  2.]
 [ 4.  5.]]
>>> print inverse(data)
[[-1.66666667  0.66666667]
 [ 1.33333333 -0.33333333]]
>>> print inverse(inverse(data))
[[ 1.  2.]
 [ 4.  5.]]
```

5. Array Basics

This chapter introduces some of the basic functions which will be used throughout the text.

Basics

Before we explore the world of image manipulation as a case-study in array manipulation, we should first define a few terms which we'll use over and over again. Discussions of arrays and matrices and vectors can get confusing due to disagreements on the nomenclature. Here is a brief definition of the terms used in this tutorial, and more or less consistently in the error messages of NumPy.

The python objects under discussion are formally called "multiarray" objects, but informally we'll just call them "array" objects or just "arrays." These are different from the array objects defined in the standard `array` module (which is designed at processing one-dimensional data such as sound files).

These array objects are arrays in the computer science meaning of homogeneous block of elements, i.e. their elements all have the same "kind" (usually, a specific kind of number, such as a 64-bit integer). This is quite different from most Python container objects, which can contain heterogeneous collections. Imposing this restriction was a necessary step for a fast implementation, and in practice, it is not a cumbersome restriction in many cases[1].

Any given array object has a rank, which is the number of "dimensions" or "axes" it has. For example, a point in 3D space [1, 2, 1] is an array of rank 1 – it has one dimension. That dimension has a *length* of 3.

As another example, the array

```
1.0 0.0 0.0
0.0 1.0 2.0
```

is an array of rank 2 (it is 2-dimensional). The first dimension has a length of 2, the second dimension has a length of 3. Because the word "dimension" has many different meanings to different folks, in general the word "axis" will be used instead. Axes are numbered just like Python list indices: they start at 0, and can also be counted from the end, so that axis -1 is the last axis of an array, axis -2 is the penultimate axis, etc.

There are two important and potentially unintuitive behaviors of NumPy arrays which take some getting used to. The first is that by default, operations on arrays are performed element-wise. This means that when adding two arrays, the resulting array has as elements the pairwise sums of the two operand arrays. This is true for all operations, including multiplication. Thus, array multiplication using the * operator will default to element-wise multiplication, not matrix multiplication as used in linear algebra. Many people will want to use arrays as linear algebra-type matrices (including their rank-1 versions, vectors). For those users, the Matrix class provides a more intuitive interface. We defer discussion of the Matrix class until later. The second behavior which will catch many users by surprise is that functions which return arrays which are simply different views at the same data will in fact *share* their data. This will be discussed at length when we have more concrete examples of what exactly this means.

Now that all of these definitions and warnings are laid out, let's see what we can do with these arrays.

---

1. As we'll also see, one valid "object kind" is a Python reference – in other words, these arrays can be made heterogeneous at the cost of speed.

Creating arrays from scratch

array() and typecodes

There are many ways to create arrays. The most basic one is the use of the `array()` function:

```
>>> a = array([1.2, 3.5, -1])
```

to make sure this worked, do:

```
>>> print a
[ 1.2  3.5 -1. ]
```

The `array(numbers, typecode=None)`[1] function takes two arguments – the first one is the values, which have to be in a Python sequence object (such as a list or a tuple). The optional second argument is the typecode of the elements. If it is omitted, as in the example above, Python tries to find the one type which can represent all the elements. Since the elements we gave our example were two floats and one integer, it chose `float' as the type of the resulting array. If one specifies the typecode, one can specify unequivocally the type of the elements – this is especially useful when, for example, one wants to make sure that an array contains floats even though in some cases all of its elements are integers:

```
>>> x,y,z = 1,2,3
>>> a = array([x,y,z])          # integers are enough for 1, 2 and 3
>>> print a
[1 2 3]
>>> a = array([x,y,z], Float)   # not the default type
>>> print a
[ 1.  2.  3.]
```

Pop Quiz: hat will be the type of an array defined as follows:

```
>>> mystery = array([1, 2.0, -3j])
```

Hint: `-3j` is an imaginary number.
Answer: try it out!

A very common mistake is to call array with a set of numbers as arguments, as in `array(1,2,3,4,5)`. This doesn't produce the expected result as soon as at least two numbers are used, because the first argument to `array()` must be the entire data for the array -- thus, in most cases, a sequence of numbers. The correct way to write the preceding invocation is most likely `array((1,2,3,4,5))`.

Possible values for the second argument to the `array` creator function (and indeed to any function which accepts a so-called typecode for arrays) are:

1. One type corresponding to single ASCII characters: `Character`.
2. One unsigned numeric type: `UnsignedInt8`, used to store numbers between 0 and 255.
3. Many signed numeric types:

1. When giving "function signatures," only the most commonly used arguments and their default values will be listed. For complete function signatures, consult the Numeric Python Reference Manual.

- Signed integer choices: `Int`, `Int0`, `Int8`, `Int16`, `Int32`, and on some platforms, `Int64` and `Int128` (their ranges depend on their size).
- Floating point choices: `Float`, `Float0`, `Float8`, `Float16`, `Float32`, `Float64`, and on some platforms, `Float128`.
- Complex number choices: `Complex`, `Complex0`, `Complex8`, `Complex16`, `Complex32`, `Complex64`, `Complex128`.

The meaning of these is as follows:

- The versions without any numbers (`Int`, `Float`, `Complex`) correspond to the `int`, `float` and `complex` datatypes in Python. They are thus long integers and double-precision floating point numbers, with a complex number corresponding to two double-precision floats.
- The versions with a number following correspond to whatever words are available on the specific platform you are using which have *at least* that many bits in them. Thus, `Int0` corresponds to the smallest integer word size available, `Int8` corresponds to the smallest integer word size available which has at least 8 bits, etc. The word sizes for the complex numbers refer to the total number of bits used by both the real and imaginary parts (in other words, the data portion of an array of N `Complex128` elements uses up the same amount of memory as the data portions of two arrays of typecode `Float64` with 2N elements).

4. One non-numeric type, `PyObject`. Arrays of typecode `PyObject` are arrays of Python references, and as such their data area can contain references to any kind of Python objects.

The last typecode deserves a little comment. Indeed, it seems to indicate that arrays can be filled with any Python objects. This appears to violate the notion that arrays are homogeneous. In fact, the typecode `PyObject` *does* allow heterogeneous arrays. However, if you plan to do numerical computation, you're much better off with a homogeneous array with a potentially “large” type than with a heterogeneous array. This is because a heterogeneous array stores references to objects, which incurs a memory cost, and because the speed of computation is much slower with arrays of `PyObject`'s than with uniform number arrays. Why does it exist, then? A very useful features of arrays is the ability to slice them, dice them, select and choose from them, etc. This feature is so nice that sometimes one wants to do the same operations with, e.g., arrays of class instances. In such cases, computation speed is not as important as convenience. Also, if the array is filled with objects which are instances of classes which define the appropriate methods, then NumPy will let you do math with those objects. For example, if one creates an object class which has an `__add__` method, then arrays (created with the `PyObject` typecode) of instances of such a class can be added together. [XXXXXX make sure that’s true!].

Multidimensional Arrays

The following example shows one way of creating multidimensional arrays:

```
>>> ma = array([[1,2,3],[4,5,6]])
>>> print ma
[[1 2 3]
 [4 5 6]]
```

The first argument to `array()` in the code above is a single list containing two lists, each containing three elements. If we wanted floats instead, we could specify, as discussed in the previous section, the optional typecode we wished:

```
>>> ma_floats = array([[1,2,3],[4,5,6]], Float)
>>> print ma_floats
[[ 1.  2.  3.]
 [ 4.  5.  6.]]
```

This array allows us to introduce the notion of `shape'. The shape of an array is the set of numbers which define its dimensions. The shape of the array `ma` defined above is 2 by 3. More precisely, all arrays have a shape attribute which is a tuple of integers. So, in this case:

```
>>> print ma.shape
(2, 3)
```

Using the earlier definitions, this is a shape of *rank* 2, where the first axis has length 2, and the seond axis has length 3. The rank of an array `A` is always equal to `len(A.shape)`.

Note that `shape` is an *attribute* of `array` objects. It is the first of several which we will see throughout this tutorial. If you're not used to object-oriented programming, you can think of attributes as "features" or "qualities" of individual arrays. The relation between an array and its shape is similar to the relation between a person and their hair color. In Python, it's called an object/attribute relation.

What if one wants to change the dimensions of an array? For now, let us consider changing the shape of an array without making it "grow." Say, for example, we want to make the 2x3 array defined above (`ma`) an array of rank 1:

```
>>> flattened_ma = reshape(ma, (6,))
>>> print flattened_ma
[1 2 3 4 5 6]
```

One can change the shape of arrays to any shape as long as the product of all the lengths of all the axes is kept constant (in other words, as long as the number of elements in the array doesn't change):

```
>>> a = array([1,2,3,4,5,6,7,8])
[1 2 3 4 5 6 7 8]
>>> print a
>>> b = reshape(a, (2,4))        # 2*4 == 8
[[1 2 3 4]
 [5 6 7 8]]
>>> print b
>>> c = reshape(b, (4,2)         # 4*2 == 8
>>> print c
[[1 2]
 [3 4]
 [5 6]
 [7 8]]
```

Notice that we used a new function, `reshape()`. It, like `array()`, is a function defined in the `Numeric` module. It expects an array as its first argument, and a shape as its second argument. The shape has to be a sequence of integers (a list or a tuple). Keep in mind that a tuple with a single element needs a comma at the end; the right shape tuple for a rank-1 array with 5 elements is `(5,)`, not `(5)`.

One nice feature of shape tuples is that one entry in the shape tuple is allowed to be `-1`. The `-1` will be automatically replaced by whatever number is needed to build a shape which does not change the size of the array. Thus:

```
>>> a = reshape(array(range(25)), (5,-1))
>>> print a, a.shape
[[ 0  1  2  3  4]
 [ 5  6  7  8  9]
 [10 11 12 13 14]
 [15 16 17 18 19]
 [20 21 22 23 24]] (5, 5)
```

The shape of an array is a modifiable attribute of the array. You can therefore change the shape of an array simply by assigning a new shape to it:

```
>>> a = array([1,2,3,4,5,6,7,8,9,10])
>>> a.shape
(10,)
>>> a.shape = (2,5)
>>> print a
[[ 1  2  3  4  5]
 [ 6  7  8  9 10]]
>>> a.shape = (10,1)             # second axis has length 1
>>> print a
[[ 1]
 [ 2]
 [ 3]
 [ 4]
 [ 5]
 [ 6]
 [ 7]
 [ 8]
 [ 9]
 [10]]
>>> a.shape = (5,-1)             # note the -1 trick described above
>>> print a
[[ 1  2]
 [ 3  4]
 [ 5  6]
 [ 7  8]
 [ 9 10]]
```

As in the rest of Python, violating rules (such as the one about which shapes are allowed) results in exceptions:

```
>>> a.shape = (6,-1)
Traceback (innermost last):
  File "<stdin>", line 1, in ?
ValueError: total size of new array must be unchanged
```

The default printing routine provided by the Numeric module prints arrays as follows:

1 The last axis is always printed left to right
2 The next-to-last axis is printed top to bottom
3 The remaining axes are printed top to bottom with increasing numbers of separators

---

This explains why rank-1 arrays are printed from left to right, rank-2 arrays have the first dimension going down the screen and the second dimension going from left to right, etc.

If you want to change the shape of an array so that it has more elements than it started with (i.e. grow it), then you have many options: One solution is to use the `concat()` method discussed later. An alternative is to use the `array()` creator function with existing arrays as arguments:

```
>>> print a
[0 1 2 3 4 5 6 6 7]
>>> b = array([a,a])
```

```
>>> print b
[[1 2 3 4 5 6 7 8]
 [1 2 3 4 5 6 7 8]]
>>> print b.shape
(2, 8)
```

XXX reshape

A final possibility is the `resize()`function, which takes a "base" array as its first argument and the desired shape as the second argument. Unlike `reshape()`, the shape argument to `resize()` can corresponds to a smaller or larger shape than the input array. Smaller shapes will result in arrays with the data at the "beginning" of the input array, and larger shapes result in arrays with data containing as many replications of the input array as are needed to fill the shape. For example, starting with a simple array

```
>>> base = array([0,1])
```

one can quickly build a large array with replicated data:

```
>>> big = resize(base, (9,9))
>>> print big
[[0 1 0 1 0 1 0 1 0]
 [1 0 1 0 1 0 1 0 1]
 [0 1 0 1 0 1 0 1 0]
 [1 0 1 0 1 0 1 0 1]
 [0 1 0 1 0 1 0 1 0]
 [1 0 1 0 1 0 1 0 1]
 [0 1 0 1 0 1 0 1 0]
 [1 0 1 0 1 0 1 0 1]
 [0 1 0 1 0 1 0 1 0]]
```

and if you imported the `view` function from the `NumTut` package, you can do:

```
>>> view(resize(base, (100,100)))
# grey grid of horizontal lines is shown
>>> view(resize(base, (101,101)))
# grey grid of alternating black and white pixels is shown
```

Sections denoted such as this one with an "eye" symbol will be used to indicate aspects of the functions which may not be needed for a first introduction at NumPy, but which should be mentioned for the sake of completeness.

The `array` constructor takes a mandatory `data` argument, an optional typecode, and an optional `copy` argument. If the `data` argument is a sequence, then array creates a new object of type multiarray, and fills the array with the elements of the `data` object. The shape of the array is determined by the size and nesting arrangement of the elements of data.

If `data` is not a sequence, then the array returned is an array of shape `()` (the empty tuple), of typecode `'O'`, containing a single element, which is `data`.

Creating arrays with values specified `on-the-fly'

zeros() and ones()

Often, one needs to manipulate arrays filled with numbers which aren't available beforehand. The Numeric module provides a few functions which create arrays from scratch:

`zeros()` and `ones()` simply create arrays of a given shape filled with zeros and ones respectively:

```
>>> z = zeros((3,3))
>>> print z
[[0 0 0]
 [0 0 0]
 [0 0 0]]
>>> o = ones([2,3])
>>> print o
[[1 1 1]
 [1 1 1]]
```

Note that the first argument is a shape – it needs to be a list or a tuple of integers. Also note that the default type for the returned arrays is `Int`, which you can feel free to override using something like:

```
>>> o = ones((2,3), Float)
>>> print o
[[ 1.  1.  1.]
 [ 1.  1.  1.]]
```

arrayrange()

The `arrayrange()` function is similar to the `range()` function in Python, except that it returns an array as opposed to a list.

```
>>> r = arrayrange(10)
>>> print r
[0 1 2 3 4 5 6 7 8 9]
```

Combining the `arrayrange()` with the `reshape()` function, we can get:

```
>>> big = reshape(arrayrange(100),(10,10))
>>> print big
[[ 0  1  2  3  4  5  6  7  8  9]
 [10 11 12 13 14 15 16 17 18 19]
 [20 21 22 23 24 25 26 27 28 29]
 [30 31 32 33 34 35 36 37 38 39]
 [40 41 42 43 44 45 46 47 48 49]
 [50 51 52 53 54 55 56 57 58 59]
 [60 61 62 63 64 65 66 67 68 69]
 [70 71 72 73 74 75 76 77 78 79]
 [80 81 82 83 84 85 86 87 88 89]
 [90 91 92 93 94 95 96 97 98 99]]
>>> view(reshape(arrayrange(10000),(100,100)))
# array of increasing lightness from top down (slowly) and from left to
# right (faster) is shown
```

`arange()` is a shorthand for `arrayrange()`.

One can set the start, stop and step arguments, which allows for more varied ranges:

```
>>> print arrayrange(10,-10,-2)
[10  8  6  4  2  0  -2  -4  -6  -8]
```

An important feature of arrayrange is that it can be used with non-integer starting points and strides:

```
>>> print arrayrange(5.0)
[ 0. 1. 2. 3. 4.]
>>> print arrayrange(0, 1, .2)
[ 0.   0.2  0.4  0.6  0.8]
```

If you want to create an array with just one value, repeated over and over, you can use the * operator applied to lists

```
>>> a = array([[3]*5]*5)
>>> print a
[[3 3 3 3 3]
 [3 3 3 3 3]
 [3 3 3 3 3]
 [3 3 3 3 3]
 [3 3 3 3 3]]
```

but that is relatively slow, since the duplication is done on Python lists. A quicker way would be to start with 0's and add 3:

```
>>> a = zeros([5,5]) + 3
>>> print a
[[3 3 3 3 3]
 [3 3 3 3 3]
 [3 3 3 3 3]
 [3 3 3 3 3]
 [3 3 3 3 3]]
```

The optional typecode argument can force the typecode of the resulting array, which is otherwise the “highest” of the starting and stopping arguments. The starting argument defaults to 0 if not specified. `arange` is a synonym for `arrayrange`. Note that if a typecode is specified which is “lower” than that which arrayrange would normally use, the array is the result of a precision-losing cast (a round-down, as that used in the `astype` method for arrays.)

Creating an array from a function: fromfunction()

Finally, one may want to create an array with contents which are the result of a function evaluation. This is done using the `fromfunction()` function, which takes two arguments, a shape and a callable object (usually a function). For example:

```
>>> def dist(x,y):
...   return (x-5)**2+(y-5)**2   # distance from point (5,5) squared
...
>>> m = fromfunction(dist, (10,10))
>>> print m
[[50 41 34 29 26 25 26 29 34 41]
 [41 32 25 20 17 16 17 20 25 32]
 [34 25 18 13 10  9 10 13 18 25]
 [29 20 13  8  5  4  5  8 13 20]
 [26 17 10  5  2  1  2  5 10 17]
 [25 16  9  4  1  0  1  4  9 16]
 [26 17 10  5  2  1  2  5 10 17]
 [29 20 13  8  5  4  5  8 13 20]
 [34 25 18 13 10  9 10 13 18 25]
```

```
 [41 32 25 20 17 16 17 20 25 32]]
>>> view(fromfunction(dist, (100,100))
# shows image which is dark in topleft corner, and lighter away from it.
>>> m = fromfunction(lambda i,j,k: 100*(i+1)+10*(j+1)+(k+1), (4,2,3))
>>> print m
[[[111 112 113]
  [121 122 123]]
 [[211 212 213]
  [221 222 223]]
 [[311 312 313]
  [321 322 323]]
 [[411 412 413]
  [421 422 423]]]
```

By examining the above examples, one can see that `fromfunction()` creates an array of the shape specified by its second argument, and with the contents corresponding to the value of the function argument (the first argument) evaluated at the indices of the array. Thus the value of `m[3,4]` in the first example above is the value of `dist` when `x=3` and `y=4`. Similarly for the lambda function in the second example, but with a rank-3 array.

The implementation of fromfunction consists of:

```
def fromfunction(function, dimensions):
    return apply(function, tuple(indices(dimensions)))
```

which means that the function function is called for each element in the sequence indices(dimensions). As described in the definition of indices, this consists of arrays of indices which will be of rank one less than that specified by dimensions. This means that the function argument must accept the same number of arguments as there are dimensions in dimensions, and that each argument will be an array of the same shape as that specified by dimensions. Furthermore, the array which is passed as the first argument corresponds to the indices of each element in the resulting array along the first axis, that which is passed as the second argument corresponds to the indices of each element in the resulting array along the second axis, etc. A consequence of this is that the function which is used with fromfunction will work as expected only if it performs a separable computation on its arguments, and expects its arguments to be indices along each axis. Thus, no logical operation on the arguments can be performed, or any non-shape preserving operation. The first example below satisfies these requirements, hence works (the `x` and `y` arrays both get 10x10 arrays as input corresponding to the values of the indices along the two dimensions), while the second array attemps to do a comparison test on an array of indices, which fails.

```
>>> def buggy(test):
...     if test > 4: return 1
...     else: return 0
...
>>> print fromfunction(buggy, (10,))
Traceback (innermost last):
  File "<stdin>", line 1, in ?
  File "C:\PYTHON\LIB\Numeric.py", line 157, in fromfunction
    return apply(function, tuple(indices(dimensions)))
  File "<stdin>", line 2, in buggy
TypeError: Comparison of multiarray objects is not implemented.
```

If you need to fill an array with the result of a function which does not meet these criteria, you can always use a function like:

```
def slowfromfunction(function, shape):
    # XXXXXX  I need to come up with a version of that...
```

identity()

The simplest array constructor is the `identity(n)` function, which takes a single integer argument and returns a square identity array of that size of integers:

```
>>> print identity(5)
[[1 0 0 0 0]
 [0 1 0 0 0]
 [0 0 1 0 0]
 [0 0 0 1 0]
 [0 0 0 0 1]]
>>> view(identity(100))
# shows black square with a single white diagonal
```

Coercion and Casting

We've mentioned the typecodes of arrays, and how to create arrays with the right typecode, but we haven't covered what happens when arrays with different typecodes interact.

Automatic Coercions and Binary Operations

The rules followed by NumPy when performing binary operations on arrays mirror those used by Python in general. Operations between numeric and non-numeric types are not allowed (e.g. an array of characters can't be added to an array of numbers), and operations between mixed number types (e.g. floats and integers, floats and omplex numbers, or in the case of NumPy, operations between any two arrays with different numeric typecodes) first perform a coercion of the 'smaller' numeric type to the type of the 'larger' numeric type. Finally, when scalars and arrays are operated on together, the scalar is converted to a rank-0 array first. Thus, adding a "small" integer to a "large" floating point array is equivalent to first casting the integer "up" to the typecode of the array:

```
>>> arange(0, 1.0, .1) + 12
array([ 12. ,  12.1,  12.2,  12.3,  12.4,  12.5,  12.6,  12.7,  12.8,
12.9])
```

The automatic coercions are described in Figure 1.

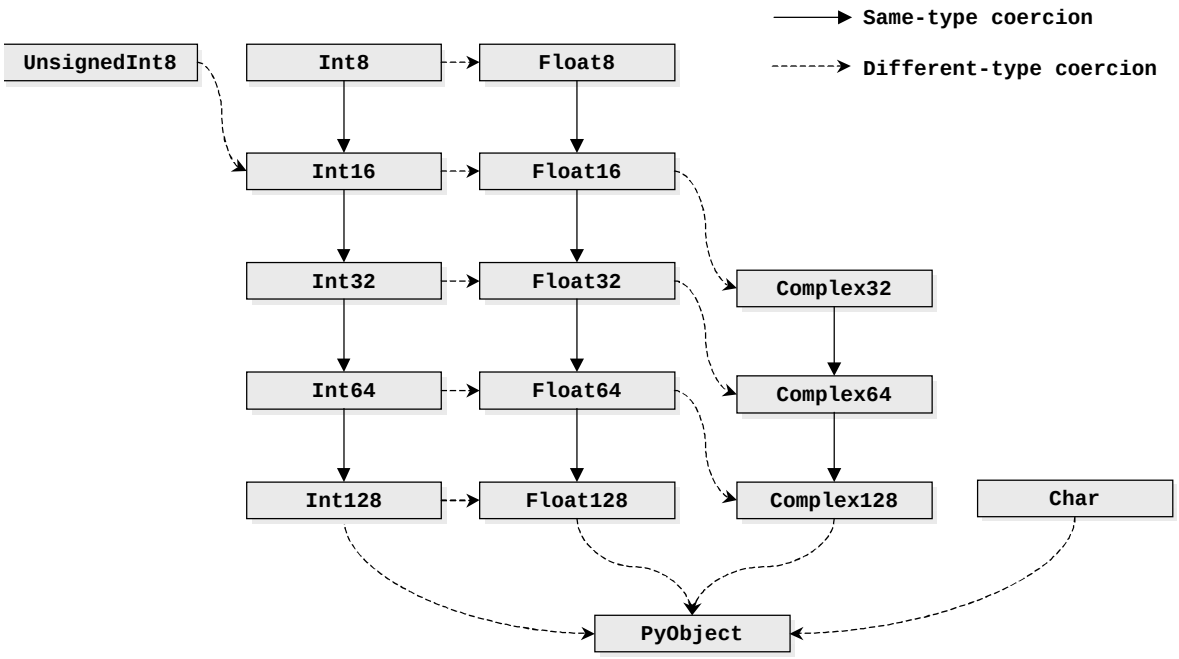

Figure 1 Up-casts are indicated with arrows. Down-casts are allowed by the `astype()` method, but may result in loss of information.

Deliberate up-casting: The asarray function

One more array constructor is the `asarray()` function. It is used if you want to have an array of a specific typecode and you don't know what typecode array you have (for example, in a generic function which can operate on all kinds of arrays, but needs them to be converted to complex arrays). If the array it gets as an argument is of the right typecode, it will get sent back unchanged. If the array is not of the right typecode, each element of the new array will be the result of the coercion to the new type of the old elements. `asarray()` will refuse to operate if there might be loss of information -- in other words, `asarray()` only casts 'up'.

`asarray` is also used when you have a function that operates on arrays, but you want to allow people to call it with an arbitrary python sequence object. This gives your function a behavior similar to that of most of the builtin functions that operate on arrays.

The typecode value table

The typecodes identifiers (`Float0`, etc.) have as values single-character strings. The mapping between typecode and character strings is machine dependent. An example of the correspondences between typecode characters and the typecode identifiers for 32-bit architectures are shown in Table 3-X.

Table 1: Typecode character/identifier table on a Pentium computer

Character	# of bytes	# of bits	Identifiers
`D`	16	128	`Complex, Complex64`
`F`	8	64	`Complex0, Complex16, Complex32, Complex8`
`d`	8	64	`Float, Float64`
`f`	4	32	`Float0, Float16, Float32, Float8`

Table 1: Typecode character/identifier table on a Pentium computer

Character	# of bytes	# of bits	Identifiers
`l`	4	32	`Int`
`1`	1	8	`Int0, Int8`
`s`	2	16	`Int16`
`i`	4	32	`Int32`

Consequences of silent upcasting

When dealing with very large arrays of floats and if precision is not important (or arrays of small integers), then it may be worthwhile to cast the arrays to "small" typecodes, such as `Int8`, `Int16` or `Float32`. As the standard Python integers and floats correspond to the typecodes `Int32` and `Float64`, using them in apparently "innocent" ways will result in up-casting, which may null the benefit of the use of small typecode arrays. For example:

```
>>> mylargearray.typecode()
'f'                              #  a.k.a. Float32 on a Pentium
>>> mylargearray.itemsize()
4
>>> mylargearray = mylargearray + 1# 1 is an Int64 on a Pentium
>>> mylargearray.typecode()      # see Fig. 1 for explanation.
'd'
>>> mylargearray.itemsize()
8
```

Note that the sizes returned by the `itemsize()` method are expressed in bytes.

To prevent this problem, one should use arrays containing a single number, with the appropriate bytecode. This can be facilitated by a few convenience functions, such as:

```
toChar = lambda x: array(x, Character)
toInt8 = lambda x: array(x, Int8)# or use variable names such as Byte
toInt16 = lambda x: array(x, Int16)
toInt32 = lambda x: array(x, Int32)
toFloat32 = lambda x: array(x, Float32)
toFloat64 = lambda x: array(x, Float64)

>>> mylargearray.typecode(), mylargearray.itemsize()
('f', 4)                         #  start again
>>> mylargearray = mylargearray + toFloat32(1)
>>> mylargearray.typecode(), mylargearray.itemsize()
('f', 4)                         #  no up-casting, no size change
```

Deliberate casts (potentially down): the astype method

You may also force NumPy to cast any number array to another number array. For example, to take an array of any numeric type (IntX or FloatX or ComplexX or UnsignedInt8) and convert it to a 64-bit float, one can do:

```
>>> floatarray = otherarray.astype(Float64)
```

The typecode can be any of the number typecodes, “larger” or “smaller". If it is larger, this is a cast-up, as if asarray() had been used. If it is smaller, the standard casting rules of the underlying language (C) are used, which means that truncation or loss of precision can occur:

```
>>> print x
[ 0.   0.4  0.8  1.2  1.6]
>>> x.astype(Int)
array([0, 0, 0, 1, 1])
```

If the typecode used with `astype()` is the original array’s typecode, then a copy of the original array is returned.

Operating on Arrays

Simple operations

If you have a keen eye, you have noticed that some of the previous examples did something new. It added a number to an array. Indeed, most Python operations applicable to numbers are directly applicable to arrays:

```
>>> print a
[1 2 3]
>>> print a * 3
[3 6 9]
>>> print a + 3
[4 5 6]
```

Note that the mathematical operators behave differently depending on the types of their operands. When one of the operands is an array and the other is a number, the number is added to all the elements of the array and the resulting array is returned. This is called *broadcasting*. This also occurs for unary mathematical operations such as sin and the negative sign

```
>>> print sin(a)
[ 0.84147098  0.90929743  0.14112001]
>>> print -a
[-1 -2 -3]
```

When both elements are arrays with the same shape, then a new array is created, where each element is the sum of the corresponding elements in the original arrays:

```
>>> print a + a
[2 4 6]
```

If the operands of operations such as addition are arrays which have the same rank but different non-integer dimensions, then an exception is generated:

```
>>> print a
[1 2 3]
>>> b = array([4,5,6,7])         # note this has four elements
>>> print a + b
Traceback (innermost last):
  File ``<stdin>``, line 1, in ?
ArrayError: frames are not aligned
```

This is because there is no reasonable way for NumPy to interpret addition of a `(3,)` shaped array and a `(4,)` shaped array.

Note what happens when adding arrays with different rank

```
>>> print a
```

```
[1 2 3]
>>> print b
[[ 4  8 12]
 [ 5  9 13]
 [ 6 10 14]
 [ 7 11 15]]
>>> print a + b
[[ 5 10 15]
 [ 6 11 16]
 [ 7 12 17]
 [ 8 13 18]]
```

This is another form of broadcasting. To understand this, one needs to look carefully at the shapes of a and b:

```
>>> a.shape
(3,)
>>> b.shape
(4,3)
```

Because array a's last dimension had length 3 and array b's last dimension also had length 3, those two dimensions were "matched" and a new dimension was created and automatically "assumed" for array a. The data already in a was "replicated" as many times as needed (4, in this case) to make the two shapes of the operand arrays conform. This replication (broadcasting) occurs when arrays are operands to binary operations and their shapes differ and when the following conditions are true:

- starting from the last axis, the axis lengths (dimensions) of the operands are compared
- if both arrays have an axis length greater than 1, an exception is raised
- if one array has an axis length greater than 1, then the other array's axis is "stretched" to match the length of the first axis -- if the other array's axis is not present (i.e., if the other array has smaller rank), then a new axis of the same length is created.

This algorithm is complex, but intuitive in practice. For more details, consult the Numeric Reference.

Getting and Setting array values

Just like other Python sequences, array contents are manipulated with the [] notation. For rank-1 arrays, there are no differences between list and array notations:

```
>>> a = arrayrange(10)
>>> print a[0]                  # get first element
0
>>> print a[1:5]                # get second through fifth element
[1 2 3 4]
>>> print a[:-1]                # get last element
9
```

The first difference with lists comes with multidimensional indexing. If an array is multidimensional (of rank > 1), then specifying a single integer index will return an array of dimension one less than the original array.

```
>>> a = arrayrange(9)
>>> a.shape = (3,3)
>>> print a
[[0 1 2]
 [3 4 5]
 [6 7 8]]
>>> print a[0]                  # get first row, not first element!
[0 1 2]
```

```
>>> print a[1]                  # get second row
[3 4 5]
```

To get to individual elements in a rank-2 array, one specifies both indices separated by commas:

```
>>> print a[0,0]                # get elt at first row, first column
0
>>> print a[0,1]                # get elt at first row, second column
1
>>> print a[1,0]                # get elt at second row, first column
3
>>> print a[2,-1]               # get elt at third row, last column
8
```

Of course, the `[]` notation can be used to *set* values as well:

```
>>> a[0,0] = 123
>>> print a
[[123   1   2]
 [  3   4   5]
 [  6   7   8]]
```

Note that when referring to rows, the right hand side of the equal sign needs to be a sequence which “fits” in the referred array subset (in the code sample below, a 3-element row):

```
>>> a[1] = [10,11,12]
>>> print a
[[123   1   2]
 [ 10  11  12]
 [  6   7   8]]
```

Slicing Arrays

The standard rules of Python slicing apply to arrays, on a per-dimension basis. Assuming a 3x3 array:

```
>>> a = reshape(arrayrange(9),(3,3))
>>> print a
[[0 1 2]
 [3 4 5]
 [6 7 8]]
```

The plain [:] operator slices from beginning to end:

```
>>> print a[:,:]
[[0 1 2]
 [3 4 5]
 [6 7 8]]
```

In other words, [:] with no arguments is the same as [:] for lists – it can be read ``all indices along this axis. So, to get the second row along the second dimension:

```
>>> print a[:,1]
[1 4 7]
```

Note that what was a “column” vector is now a “row” vector -- any “integer slice” (as in the 1 in the example above) results in a returned array with rank one less than the input array.

If one does not specify as many slices as there are dimensions in an array, then the remaining slices are assumed to be ``all''. If A is a rank-3 array, then

```
A[1] == A[1,:] == A[1,:,:]
```

There is one addition to the slice notation for arrays which does not exist for lists, and that is the optional third argument, meaning the ``step size" also called stride or increment. Its default value is 1, meaning return every element in the specified range. Alternate values allow one to skip some of the elements in the slice:

```
>>> a = arange(12)
>>> print a
[ 0  1  2  3  4  5  6  7  8  9 10 11]
>>> print a[::2]                # return every *other* element
[ 0  2  4  6  8 10]
```

Negative strides are allowed as long as the starting index is greater than the stopping index:

```
>>> a = reshape(arrayrange(9),(3,3))
>>> print a
[[0 1 2]
 [3 4 5]
 [6 7 8]]
>>> print a[:, 0]
[0 3 6]
>>> print a[0:3, 0]
[0 3 6]
>>> print a[2:-1, 0]
[6 3 0]
```

If a negative stride is specified and the starting or stopping indices are omitted, they default to “end of axis” and “beginning of axis” respectively. Thus, the following two statements are equivalent for the array given:

```
>>> print a[2:-1, 0]
[6 3 0]
>>> print a[::-1, 0]
[6 3 0]
>>> print a[::-1]               # this reverses only the first axis
[[6 7 8]
 [3 4 5]
 [0 1 2]]
>>> print a[::-1,::-1]          # this reverses both axes
[[8 7 6]
 [5 4 3]
 [2 1 0]]
```

One final way of slicing arrays is with the keyword `...` This keyword is somewhat complicated. It stands for ``however many `:' I need depending on the rank of the object I'm indexing, so that the indices I *do* specify are at the end of the index list as opposed to the usual beginning.``

So, if one has a rank-3 array `A`, then `A[...,0]` is the same thing as `A[:,:,0]` but if `B` is rank-4, then `B[...,0]` is the same thing as: `B[:,:,:,0]`. Only one `...` is expanded in an index expression, so if one has a rank-5 array `C`, then: `C[...,0,...]` is the same thing as `C[:,:,:,0,:]`.

6. Ufuncs

What are Ufuncs?

The operations on arrays that were mentioned in the previous section (element-wise addition, multiplication, etc.) all share some features -- they all follow similar rules for broadcasting, coercion and “element-wise operation". Just like standard addition is available in Python through the add function in the operator module, array operations are available through callable objects as well. Thus, the following objects are available in the Numeric module:

Table 2: Universal Functions, or `ufunc`s. The operators which invoke them when applied to arrays are indicated in parentheses. The entries in slanted typeface refer to unary ufuncs, while the others refer to binary ufuncs.

add (+)	subtract (-)	multiply (*)	divide (/)
remainder (%)	power (**)	*arccos*	*arccosh*
arcsin	*arcsinh*	*arctan*	*arctanh*
cos	*cosh*	*exp*	*log*
log10	*sin*	*sinh*	*sqrt*
tan	*tanh*	maximum	minimum
conjugate	equal (==)	not_equal (!=)	greater (>)
greater_equal (>=)	less (<)	less_equal (<=)	logical_and (and)
logical_or (or)	logical_xor	logical_not (not)	bitwise_and (&)
bitwise_or (|)	bitwise_xor	bitwise_not (~)	

All of these ufuncs can be used as functions. For example, to use `add`, which is a binary ufunc (i.e. it takes two arguments), one can do either of:

```
>>> a = arange(10)
>>> print add(a,a)
[ 0  2  4  6  8 10 12 14 16 18]
>>> print a + a
[ 0  2  4  6  8 10 12 14 16 18]
```

In other words, the + operator on arrays performs exactly the same thing as the `add` ufunc when operated on arrays. For a unary ufunc such as `sin`, one can do, e.g.:

```
>>> a = arange(10)
>>> print sin(a)
[ 0.          0.84147098  0.90929743  0.14112001 -0.7568025  -0.95892427
      -0.2794155   0.6569866   0.98935825  0.41211849]
```

Unary ufuncs return arrays with the same shape as their arguments, but with the contents corresponding to the corresponding mathematical function applied to each element (sin(0)=0, sin(1)=0.84147098, etc.).

There are three additional features of ufuncs which make them different from standard Python functions. They can operate on any Python sequence in addition to arrays; they can take an "output" argument; they have attributes which are themselves callable with arrays and sequences. Each of these will be described in turn.

Ufuncs can operate on any Python sequence

Ufuncs have so far been described as callable objects which take either one or two arrays as arguments (depending on whether they are unary or binary). In fact, any Python sequence which can be the input to the array() constructor can be used. The return value from ufuncs is always an array. Thus:

```
>>> add([1,2,3,4], (1,2,3,4))
array([2, 4, 6, 8])
```

Ufuncs can take output arguments

In many computations with large sets of numbers, arrays are often used only once. For example, a computation on a large set of numbers could involve the following step

```
dataset = dataset * 1.20
```

This operation as written needs to create a temporary array to store the results of the computation, and then eventually free the memory used by the original dataset array (provided there are no other references to the data it contains). It is more efficient, both in terms of memory and computation time, to do an "in-place" operation. This can be done by specifying an existing array as the place to store the result of the ufunc. In this example, one can write:

```
multiply(dataset, 1.20, dataset)
```

This is not a step to take lightly, however. For example, the "big and slow" version (`dataset = dataset * 1.20`) and the "small and fast" version above will yield different results in two cases:

- If the typecode of the target array is not that which would normally be computed, the operation will fail and raise a TypeError exception.
- If the target array corresponds to a different "view" on the same data as either of the source arrays, inconsistencies will result. For example,

  ```
  >>> a = arange(5, typecode=Float64)
  >>> print a[::-1] * 1.2
  [ 4.8  3.6  2.4  1.2  0. ]
  >>> multiply(a[::-1], 1.2, a)
  array([ 4.8 ,  3.6 ,  2.4 ,  4.32,  5.76])
  >>> print a
  [ 4.8   3.6   2.4   4.32  5.76]
  ```

  This is because the ufunc does not know which arrays share which data, and in this case the overwriting of the data contents follows a different path through the shared data space of the two arrays, thus resulting in strangely distorted data.

Ufuncs have special methods

The reduce ufunc method

If you don't know about the `reduce` command in Python, review section 5.1.1 of the Python Tutorial (*http://www.python.org/doc/tut/functional.html*). Briefly, `reduce` is most often used with two arguments, a callable object (such as a function), and a sequence. It calls the callable object with the first two element of the sequence, then with the result of that operation and the third element, and so on, returning at the end the successive "reduction" of the specified callable object over the sequence elements. Similarly, the `reduce` method of ufuncs is called with a sequence as an argument, and performs the reduction of that ufunc on the sequence. As an example, adding all of the elements in a rank-1 array can be done with:

```
>>> a = array([1,2,3,4])
>>> print add.reduce(a)
10
```

When applied to arrays which are of rank greater than one, the reduction proceeds by default along the first axis:

```
>>> b = array([[1,2,3,4],[6,7,8,9]])
>>> print b
[[1 2 3 4]
 [6 7 8 9]]
>>> print add.reduce(b)
[ 7  9 11 13]
```

A different axis of reduction can be specified with a second integer argument:

```
>>> print b
[[1 2 3 4]
 [6 7 8 9]]
>>> print add.reduce(b, 1)
[10 30]
```

The accumulate ufunc method

The `accumulate` ufunc method is simular to `reduce`, except that it returns an array containing the intermediate results of the reduction:

```
>>> a = arange(10)
>>> print a
[0 1 2 3 4 5 6 7 8 9]
>>> print add.accumulate(a)
[ 0  1  3  6 10 15 21 28 36 45]  # 0, 0+1, 0+1+2, 0+1+2+3, ... 0+...+9
>>> print add.reduce(a)
45                               # same as add.accumulate(...)[-1]
```

The outer ufunc method

The third ufunc method is `outer`, which takes two arrays as arguments and returns the "outer ufunc" of the two arguments. Thus the `outer` method of the `multiply` ufunc, results in the outer product. The outer method is only supported for binary methods.

```
>>> print a
[0 1 2 3 4]
>>> print b
[0 1 2 3]
>>> print add.outer(a,b)
[[0 1 2 3]
 [1 2 3 4]
 [2 3 4 5]
 [3 4 5 6]
 [4 5 6 7]]
>>> print multiply.outer(b,a)
[[ 0  0  0  0  0]
 [ 0  1  2  3  4]
 [ 0  2  4  6  8]
 [ 0  3  6  9 12]]
>>> print power.outer(a,b)
[[ 1  0  0  0]
 [ 1  1  1  1]
```

```
[ 1  2  4  8]
[ 1  3  9 27]
[ 1  4 16 64]]
```

The reduceat ufunc method

The final ufunc method is the `reduceat` method, which I'd love to explain it, but I don't understand it (XXX).

Ufuncs always return new arrays

Except when the 'output' argument are used as described above, ufuncs always return new arrays which do not share any data with the input array.

Which are the Ufuncs?

Table 1 lists all the ufuncs. We will first discuss the mathematical ufuncs, which perform operations very similar to the functions in the `math` and `cmath` modules, albeit elementwise, on arrays. These come in two forms, unary and binary:

Unary Mathematical Ufuncs (take only one argument)

The following ufuncs apply the predictable functions on their single array arguments, one element at a time: `arccos`, `arccosh`, `arcsin`, `arcsinh`, `arctan`, `arctanh`, `cos`, `cosh`, `exp`, `log`, `log10`, `sin`, `sinh`, `sqrt`, `tan`, `tanh`.

As an example:

```
>>> print x
[0 1 2 3 4]
>>> print cos(x)
[ 1.          0.54030231 -0.41614684 -0.9899925  -0.65364362]
>>> print arccos(cos(x))
[ 0.          1.          2.          3.          2.28318531]
# not a bug, but wraparound: 2*pi%4 is 2.28318531
```

The `conjugate` ufunc takes an array of complex numbers and returns the array with entries which are the complex conjugates of the entries in the input array. If it is called with real numbers, a copy of the array is returned unchanged.

Binary Mathematical Ufuncs

These ufuncs take two arrays as arguments, and perform the specified mathematical operation on them, one pair of elements at a time: `add`, `subtract`, `multiply`, `divide`, `remainder`, `power`.

Logical Ufuncs

The ``logical" ufuncs also perform their operations on arrays in elementwise fashion, just like the ``mathematical" ones.

Two are special (`maximum` and `miminum`) in that they return arrays with entries taken from their input arrays:

```
>>> print x
[0 1 2 3 4]
>>> print y
[ 2.   2.5  3.   3.5  4. ]
>>> print maximum(x, y)
[ 2.   2.5  3.   3.5  4. ]
>>> print minimum(x, y)
```

```
[ 0.  1.  2.  3.  4.]
```

The others all return arrays of 0's or 1's: `equal`, `not_equal`, `greater`, `greater_equal`, `less`, `less_equal`, `logical_and`, `logical_or`, `logical_xor`, `logical_not`, `bitwise_and`, `bitwise_or`, `bitwise_xor`, `bitwise_not`.

These are fairly self-explanatory, especially with the associated symbols from the standard Python version of the same operations in Table 1 above. The `logical_*` ufuncs perform their operations (and, or, etc.) using the truth value of the elements in the array (equality to 0 for numbers and the standard truth test for PyObject arrays). The `bitwise_*` ufuncs, on the other hand, can be used only with integer arrays (of any word size), and will return integer arrays of the larger bit size of the two input arrays:

```
>>> x
array([7, 7, 0],'1')
>>> y
array([4, 5, 6])
>>> bitwise_and(x,y)
array([4, 5, 0],'i')
```

We've already discussed how to find out about the contents of arrays based on the indices in the arrays – that's what the various slice mechanisms are for. Often, especially when dealing with the result of computations or data analysis, one needs to ``pick out'' parts of matrices based on the content of those matrices. For example, it might be useful to find out which elements of an array are negative, and which are positive. The comparison ufuncs are designed for just this type of operation. Assume an array with various positive and negative numbers in it (for the sake of the example we'll generate it from scratch):

```
>>> print a
[[ 0  1  2  3  4]
 [ 5  6  7  8  9]
 [10 11 12 13 14]
 [15 16 17 18 19]
 [20 21 22 23 24]]
>>> b = sin(a)
>>> print b
[[ 0.          0.84147098  0.90929743  0.14112001 -0.7568025 ]
 [-0.95892427 -0.2794155   0.6569866   0.98935825  0.41211849]
 [-0.54402111 -0.99999021 -0.53657292  0.42016704  0.99060736]
 [ 0.65028784 -0.28790332 -0.96139749 -0.75098725  0.14987721]
 [ 0.91294525  0.83665564 -0.00885131 -0.8462204  -0.90557836]]
>>> print less_equal(b, 0)
[[1 0 0 0 1]
 [1 1 0 0 0]
 [1 1 1 0 0]
 [0 1 1 1 0]
 [0 0 1 1 1]]
```

This last example has 1's where the corresponding elements are less than or equal to 0, and 0's everywhere else.

```
>>> view(greater(greeceBW, .3))
# shows a binary image with white where the pixel value was greater than
.3
```

Ufunc shorthands

`Numeric` defines a few functions which correspond to often-used uses of ufuncs: for example, `add.reduce()` is synonymous with the `sum()` utility function:

```
>>> a = arange(5)                # [0 1 2 3 4]
```

```
>>> print sum(a)                    # 0 + 1 + 2 + 3 + 4
10
```

Similarly, `cumsum` is equivalent to `add.accumulate` (for ``cumulative sum``), `product` to `multiply.reduce`, and `cumproduct` to `multiply.accumulate`.

Additional ``utility'' functions which are often useful are `alltrue` and `sometrue`, which are defined as `logical_and.reduce` and `logical_or.reduce` respectively:

```
>>> a = array([0,1,2,3,4])
>>> print greater(a,0)
[0 1 1 1 1]
>>> alltrue(greater(a,0))
0
>>> sometrue(greater(a,0))
1
```

7. Pseudo Indices

Tbis chapter discusses pseudo-indices, which allow arrays to have their shapes modified by adding axes, sometimes only for the duration of the evaluation of a Python expression.

Consider multiplication of a rank-1 array by a scalar:

```
>>> a = array([1,2,3])
>>> a * 2
[2 4 6]
```

This should be trivial to you by now. We've just multiplied a rank-1 array by a scalar (which is converted to a rank-0 array). In other words, the rank-0 array was broadcast to the next rank. This works for adding some two rank-1 arrays as well:

```
>>> print a
[1 2 3]
>>> a + array([4])
[5 6 7]
```

but it won't work if either of the two rank-1 arrays have non-matching dimensions which aren't 1 – put another way, broadcast only works for dimensions which are either missing (e.g. a lower-rank array) or for dimensions of 1.

With this in mind, consider a classic task, matrix multiplication. Suppose we want to multiply the row vector [10,20] by the column vector [1,2,3].

```
>>> a = array([10,20])
>>> b = array([1,2,3])
>>> a * b
Traceback (innermost last):
  File "<stdin>", line 1, in ?
ValueError: frames are not aligned example
```

This makes sense – we're trying to multiply a rank-1 array of shape (2,) with a rank-1 array of shape (3,). This violates the laws of broadcast. What we really want to do is make the second vector a vector of shape (3,1), so that the first vector can be broadcast accross the second axis of the second vector. One way to do this is to use the reshape function:

```
>>> a.shape
(2,)
>>> b.shape
(3,)
>>> b2 = reshape(b, (3,1))
>>> print b2
[[1]
 [2]
 [3]]
>>> b2.shape
(3, 1)
>>> print a * b2
[[10 20]
```

```
[20 40]
[30 60]]
```

This is such a common operation that a special feature was added (it turns out to be useful in many other places as well) – the `NewAxis` ``pseudo-index'', originally developed in the Yorick language. `NewAxis` is an index, just like integers, so it is used inside of the slice brackets []. It can be thought of as meaning ``add a new axis here,'' in much the same ways as adding a 1 to an array's shape adds an axis. Again, examples help clarify the situation:

```
>>> print b
[1 2 3]
>>> b.shape
(3,)
>>> c = b[:, NewAxis]
>>> print c
[[1]
 [2]
 [3]]
>>> c.shape
(3,1)
```

Why use such a pseudo-index over the reshape function or shape assignments? Often one doesn't really want a new array with a new axis, one just wants it for an intermediate computation. Witness the array multiplication mentioned above, without and with pseudo-indices:

```
>>> without = a * reshape(b, (3,1))
>>> with = a * b[:,NewAxis]
```

The second is much more readable (once you understand how `NewAxis` works), and it's much closer to the intended meaning. Also, it's independent of the dimensions of the array b You might counter that using something like `reshape(b, (-1,1))` is also dimension-independent, but 1) would you argue that it's as readable? 2) how would you deal with rank-3 or rank-N arrays? The `NewAxis`-based idiom also works nicely with higher rank arrays, and with the `...` ``rubber index'' mentioned earlier. Adding an axis before the last axis in an array can be done simply with:

```
>>> a[...,NewAxis,:]
```

8. Array Functions

Most of the useful manipulations on arrays are done with functions. This might be surprising given Python's object-oriented framework, and that many of these functions could have been implemented using methods instead. Choosing functions means that the same procedures can be applied to arbitrary python sequences, not just to arrays. For example, while `transpose([[1,2],[3,4]])` works just fine, `[[1,2],[3,4]].transpose()` can't work. This approach also allows uniformity in interface between functions defined in the Numeric Python system, whether implemented in C or in Python, and functions defined in extension modules. The use of array methods is limited to functionality which depends critically on the implementation details of array objects. Array methods are discussed in the next chapter.

We've already covered two functions which operate on arrays, `reshape` and `resize`.

take(a, indices, axis=0)

`take` is in some ways like the slice operations. It selects the elements of the array it gets as first argument based on the indices it gets as a second argument. Unlike slicing, however, the array returned by `take` has the same rank as the input array. This is again much easier to understand with an illustration:

```
>>> print a
[[ 0  1  2  3  4]
 [ 5  6  7  8  9]
 [10 11 12 13 14]
 [15 16 17 18 19]]
>>> print take(a, (0,))          # first row
[ [0 1 2 3 4]]
>>> print take(a, (0,1))         # first and second row
[[0 1 2 3 4]
 [5 6 7 8 9]]
>>> print take(a, (0,-1))        # first and last row
[[ 0  1  2  3  4]
 [15 16 17 18 19]]
```

The optional third argument specifies the axis along which the selection occurs, and the default value (as in the examples above) is 0, the first axis. If you want another axis, then you can specify it:

```
>>> print take(a, (0,), 1)       # first column
[[ 0]
 [ 5]
 [10]
 [15]]
>>> print take(a, (0,1), 1)      # first and second column
[[ 0  1]
 [ 5  6]
 [10 11]
 [15 16]]
>>> print take(a, (0,-1), 1)     # first and last column
[[ 0  4]
 [ 5  9]
 [10 14]
 [15 19]]
```

This is considered to be a ``structural'' operation, because its result does not depend on the content of the arrays or the result of a computation on those contents but uniquely on the structure of the array. Like all such structural operations, the default axis is 0 (the first rank). I mention it here because later in this tutorial, we will see functions which have a default axis of -1.

Take is often used to create multidimensional arrays with the indices from a rank-1 array. As in the earlier examples, the shape of the array returned by `take()` is a combination of the shape of its first argument and the shape of the array that elements are "taken" from -- when that array is rank-1, the shape of the returned array has the same shape as the index sequence. [XXX vague]

```
>>> x = arange(10) * 100
>>> print x
[  0 100 200 300 400 500 600 700 800 900]
>>> print take(x, [[2,4],[1,2]])
[[200 400]
 [100 200]]
```

A typical example of using `take()` is to replace the grey values in an image according to a "translation table". For example, let's consider a brightening of a greyscale image. The `view()` function defined in the NumTut package automatically scales the input arrays to use the entire range of grey values, except if the input arrays are of typecode `'b'` unsigned bytes -- thus to test this brightening function, we'll first start by converting the greyscale floating point array to a greyscale byte array:

```
>>> BW = (greeceBW*256).astype('b')
>>> view(BW)                    # shows black and white picture
```

We then create a table mapping the integers 0-255 to integers 0-255 using a "compressive nonlinearity":

```
>>> table = (255- arange(256)**2 / 256).astype('b')
>>> view(table)                 # shows the conversion curve
```

To do the "taking" into an array of the right kind, we first create a blank image array with the same shape and typecode as the original array:

```
>>> BW2 = zeros(BW.shape, BW.typecode())
```

and then perform the take() operation

```
>>> BW2.flat[:] = take(table, BW.flat)
>>> view(BW2)
```

transpose(a, axes=None)

`transpose` takes an array and returns a new array which corresponds to a with the order of axes specified by the second argument. The default corresponds to flipping the order of all the axes (it is equivalent to `a.shape[::-1]` if a is the input array).

```
>>> print a
[[ 0  1  2  3  4]
 [ 5  6  7  8  9]
 [10 11 12 13 14]
 [15 16 17 18 19]]
>>> print transpose(a)
[[ 0  5 10 15]
 [ 1  6 11 16]
 [ 2  7 12 17]
 [ 3  8 13 18]
 [ 4  9 14 19]]
>>> greece.shape                # it's a 355x242 RGB picture
```

```
(355, 242, 3)
>>> view(greece)
# picture of greek street is shown
>>> view(transpose(greece, (1,0,2)))# swap x and y, not color axis!
# picture of greek street is shown sideways
```

repeat(a, repeats, axis=0)

`repeat` takes an array and returns an array with each element in the input array repeated as often as indicated by the corresponding elements in the second array. It operates along the specified axis. So, to stretch an array evenly, one needs the repeats array to contain as many instances of the integer scaling factor as the size of the specified axis:

```
>>> view(repeat(greece, 2*ones(greece.shape[0]))) # double in X
>>> view(repeat(greece, 2*ones(greece.shape[1]), 1)) # double in Y
```

choose(a, (b0, ..., bn))

a is an array of integers between 0 and n. The resulting array will have the same shape as a, with element selected from b0,...,bn as indicating by the value of the corresponding element in a.

Assume a is an array `a` that you want to ``clip'' so that no values are greater than 100.0.

```
>>> choose(greater(a, 100.0), (a, 100.0))
```

Everywhere that greater(a, 100.0) is false (ie. 0) this will ``choose'' the corresponding value in a. Everywhere else it will ``choose'' 100.0.

This works as well with arrays. Try to figure out what the following does:

```
>>> ret = choose(greater_than(a,b), (c,d))
```

ravel(a)

returns the argument array `a` as a 1d array. It is equivalent to `reshape(a, (-1,))` or `a.flat`. Unlike `a.flat`, however, `ravel` works with non-contiguous arrays.

```
>>> print x
[[ 0  1  2  3]
 [ 5  6  7  8]
 [10 11 12 13]]
>>> x.iscontiguous()
0
>>> x.flat
Traceback (innermost last):
  File "<stdin>", line 1, in ?
ValueError: flattened indexing only available for contiguous array
>>> ravel(x)
array([ 0,  1,  2,  3,  5,  6,  7,  8, 10, 11, 12, 13])
```

nonzero(a)

nonzero() returns an array containing the indices of the elements in a that are nonzero. These indices only make sense for 1d arrays, so the function refuses to act on anything else. As of 1.0a5 this function does not work for complex arrays.

where(condition, x, y)

where(condition,x,y) returns an array shaped like condition and has elements of x and y where condition is respectively true or false

compress(condition, a, axis=0)

returns those elements of a corresponding to those elements of condition that are nonzero. condition must be the same size as the given axis of a.

```
>>> print x
[0 1 2 3]
>>> print greater(x, 2)
[0 0 0 1]
>>> print compress(greater(x, 2), x)
[3]
```

diagonal(a, k=0)

returns the entries along the k th diagonal of a (k is an offset from the main diagonal). This is designed for 2d arrays. For larger arrays, it will return the diagonal of each 2d sub-array.

```
>>> print x
[[ 0  1  2  3  4]
 [ 5  6  7  8  9]
 [10 11 12 13 14]
 [15 16 17 18 19]
 [20 21 22 23 24]]
>>> print diagonal(x)
[ 0  6 12 18 24]
>>> print diagonal(x, 1)
[ 1  7 13 19]
>>> print diagonal(x, -1)
[ 5 11 17 23]
```

trace(a, k=0)

returns the sum of the elements in a along the k th diagonal.

```
>>> print x
[[ 0  1  2  3  4]
 [ 5  6  7  8  9]
 [10 11 12 13 14]
 [15 16 17 18 19]
 [20 21 22 23 24]]
>>> print trace(x)              # 0 + 6 + 12 + 18 + 24
60
>>> print trace(x, -1)          # 5 + 11 + 17 + 23
56
>>> print trace(x, 1)           # 1 + 7 + 13 + 19
40
```

searchsorted(a, values)

Called with a rank-1 array sorted in ascending order, `searchsorted()` will return the indices of the positions in a where the corresponding values would fit.

```
>>> print bin_boundaries
[ 0.   0.1  0.2  0.3  0.4  0.5  0.6  0.7  0.8  0.9  1. ]
>>> print data
[ 0.3029573   0.79585496  0.82714031  0.77993884  0.55069605  0.76043182
       0.28511823  0.29987358  0.40286206  0.68617903]
>>> print searchsorted(bin_boundaries, data)
[4 8 9 8 6 8 3 3 5 7]
```

This can be used for example to write a simple histogramming function:

```
>>> def histogram(a, bins):
...   n = searchsorted(sort(a), bins)
...   n = concatenate([n, [len(a)]])
...   return n[1:]-n[:-1]
...
>>> print histogram([0,0,0,0,0,0,0,.33,.33,.33], arange(0,1.0,.1))
[7 0 0 3 0 0 0 0 0 0]
>>> print histogram(sin(arange(0,10,.2)), arange(-1.2, 1.2, .1))
[0 0 4 2 2 2 0 2 1 2 1 3 1 3 1 3 2 3 2 3 4 9 0 0]
```

sort(a, axis=-1)

This function returns an array containing a copy of the data in `a`, with the same shape as `a`, but with the order of the elements along the specified axis sorted. The shape of the returned array is the same as `a`'s. Thus, `sort(a, 3)` will be an array of the same shape as a, where the elements of a have been sorted along the fourth axis.

```
>>> print data
[[5 0 1 9 8]
 [2 5 8 3 2]
 [8 0 3 7 0]
 [9 6 9 5 0]
 [9 0 9 7 7]]
>>> print sort(data)             # Axis -1 by default
[[0 1 5 8 9]
 [2 2 3 5 8]
 [0 0 3 7 8]
 [0 5 6 9 9]
 [0 7 7 9 9]]
>>> print sort(data, 0)
[[2 0 1 3 0]
 [5 0 3 5 0]
 [8 0 8 7 2]
 [9 5 9 7 7]
 [9 6 9 9 8]]
```

argsort(a, axis=-1)

`argsort` will return the indices of the elements of a needed to produce `sort(a)`. In other words, for a rank-1 array, `take(a, argsort(a)) == sort(a)`.

```
>>> print data
[5 0 1 9 8]
>>> print sort(data)
[0 1 5 8 9]
>>> print argsort(data)
[1 2 0 4 3]
```

```
>>> print take(data, argsort(data))
[0 1 5 8 9]
```

argmax(a, axis=-1), argmin(a, axis=-1)

The `argmax()` function returns an array with the arguments of the maximum values of its input array a along the given axis. The returned array will have one less dimension than a. `argmin()` is just like `argmax()`, except that it returns the indices of the minima along the given axis.

```
>>> print data
[[9 6 1 3 0]
 [0 0 8 9 1]
 [7 4 5 4 0]
 [5 2 7 7 1]
 [9 9 7 9 7]]
>>> print argmax(data)
[0 3 0 2 0]
>>> print argmax(data, 0)
[0 4 1 1 4]
>>> print argmin(data)
[4 0 4 4 2]
>>> print argmin(data, 0)
[1 1 0 0 0]
```

fromstring(string, typecode)

Will return the array formed by the binary data given in string of the specified typecode. This is mainly used for reading binary data to and from files, it can also be used to exchange binary data with other modules that use python strings as storage (*e.g.* PIL). Note that this representation is dependent on the byte order. To find out the byte ordering used, use the `byteswapped()` method described on page 53.

dot(m1, m2)

The `dot()` function returns the dot product of `m1` and `m2`. This is equivalent to matrix multiply for rank-2 arrays (without the transpose). Somebody who does more linear algebra really needs to do this function right some day!

matrixmultiply(m1, m2)

The `matrixmultiply()` function is..

XXX

clip(m, m_min, m_max)

The clip function creates an array with the same shape and typecode as m, but where every entry in m that is less than m_min is replaced by m_min, and every entry greater than m_max is replaced by m_max. Entries within the range [m_min, m_max] are left unchanged.

```
>>> a = arange(9, Float)
>>> clip(a, 1.5, 7.5)
1.5000 1.5000 2.0000 3.0000 4.0000 5.0000 6.0000 7.0000 7.5000
```

indices(shape, typecode=None)

The indices function returns an array corresponding to the shape given. The array returned is an array of a new shape which is based on the specified shape, but has an added dimension of length the number of dimensions in the specified shape. For example, if the shape specified by the `shape` argument is (3,4), then the shape of the array returned will be (2,3,4) since the length of (3,4) is 2. The contents of the returned arrays are such that the ith subarray (along index 0, the first dimension) contains the indices for that axis of the elements in the array. An example makes things clearer:

```
>>> i = indices((4,3))
>>> i.shape
(2, 4, 3)
>>> print i[0]
[[0 0 0]
 [1 1 1]
 [2 2 2]
 [3 3 3]]
>>> print i[1]
[[0 1 2]
 [0 1 2]
 [0 1 2]
 [0 1 2]]
```

So, `i[0]` has an array of the specified shape, and each element in that array specifies the index of that position in the subarray for axis 0. Similarly, each element in the subarray in `i[1]` contains the index of that position in the subarray for axis 1.

swapaxes(a, axis1, axis2)

Returns a new array which shares the data of `a`, but which has the two axes specified by `axis1` and `axis2` swapped. If `a` is of rank 0 or 1, swapaxes simply returns a new reference to `a`.

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

concatenate((a0, a1, ... , an), axis=0)

Returns a new array containing copies of the data contained in all arrays `a0 ... an`. The arrays ai will be concatenated along the specified axis (0 by default). All arrays ai must have the same shape along every axis except for the one given. To concatenate arrays along a newly created axis, you can use `array((a0, ..., an))` as long as all arrays have the same shape.

```
>>> print x
[[ 0  1  2  3]
 [ 5  6  7  8]
 [10 11 12 13]]
>>> print concatenate((x,x))
[[ 0  1  2  3]
 [ 5  6  7  8]
 [10 11 12 13]
 [ 0  1  2  3]
 [ 5  6  7  8]
 [10 11 12 13]]
>>> print concatenate((x,x), 1)
[[ 0  1  2  3  0  1  2  3]
 [ 5  6  7  8  5  6  7  8]
 [10 11 12 13 10 11 12 13]]
>>> print array((x,x))
[[[ 0  1  2  3]
  [ 5  6  7  8]
  [10 11 12 13]]
 [[ 0  1  2  3]
  [ 5  6  7  8]
  [10 11 12 13]]]
```

innerproduct(a, b)

array_repr()

See section on Textual Representations of arrays.

array_str()

See section on Textual Representations of arrays.

resize(a, new_shape)

The `resize` function takes an array and a shape, and returns a new array with the specified shape, and filled with the data in the input array. Unlike the `reshape` function, the new shape does not have to yield the same size as the original array. If the new size of is less than that of the input array, the returned array contains the appropriate data from the “beginning” of the old array. If the new size is greater than that of the input array, the data in the input array is repeated as many times as needed to fill the new array.

```
>>> x = arange(10)
>>> y = resize(x, (4,2))          # note that 4*2 < 10
>>> print x
[0 1 2 3 4 5 6 7 8 9]
>>> print y
[[0 1]
 [2 3]
 [4 5]
 [6 7]]
```

```
>>> print resize(array((0,1)), (5,5))# note that 5*5 > 2
[[0 1 0 1 0]
 [1 0 1 0 1]
 [0 1 0 1 0]
 [1 0 1 0 1]
 [0 1 0 1 0]]
```

diagonal(a, offset=0, axis1=-2, axis2=-1)

The diagonal function takes an array a, and returns an array of rank 1 containing all of the elements of a such that the difference between their indices along the specified axes is equal to the specified offset. With the default values, this corresponds to all of the elements of the diagonal of a along the last two axes. *Currently this is broken for offsets other than -1, 0 and 1, and for non-square arrays.*

repeat

convolve

where(condition, x, y)

identity(n)

The identity function returns an n by n array where the diagonal elements are 1, and the off-diagonal elements are 0.

```
>>> print identity(5)
[[1 0 0 0 0]
 [0 1 0 0 0]
 [0 0 1 0 0]
 [0 0 0 1 0]
 [0 0 0 0 1]]
```

sum(a, index=0)

The sum function is a synonym for the reduce method of the add ufunc. It returns the sum of all of the elements in the sequence given along the specified axis (first axis by default).

```
>>> print x
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]
 [12 13 14 15]
 [16 17 18 19]]
>>> print sum(x)
[40 45 50 55]                   # 0+4+8+12+16, 1+5+9+13+17,
2+6+10+14+18, ...
>>> print sum(x, 1)
[ 6 22 38 54 70]                # 0+1+2+3, 4+5+6+7, 8+9+10+11, ...
```

cumsum(a, index=0)

The `cumsum` function is a synonym for the `accumulate` method of the `add` ufunc.

product(a, index=0)

The `product` function is a synonym for the `reduce` method of the `multiply` ufunc.

cumproduct(a, index=0)

The `cumproduct` function is a synonym for the `accumulate` method of the `multiply` ufunc.

alltrue(a, index=0)

The `alltrue` function is a synonym for the `reduce` method of the `logical_and` ufunc.

sometrue(a, index=0)

The `sometrue` function is a synonym for the `reduce` method of the `logical_or` ufunc.

9. Array Methods

As we discussed at the beginning of the last chapter, there are very few array methods for good reasons, and these all depend on the the implementation details. They're worth knowing, though:

itemsize()

The itemsize() method applied to an array returns the number of bytes used by any one of its elements.

```
>>> a = arange(10)
>>> a.itemsize()
4
>>> a = array([1.0])
>>> a.itemsize()
8
>>> a = array([1], Complex)
>>> a.itemsize()
16
```

iscontiguous()

Calling an array's iscontiguous() method returns true if the memory used by A is contiguous. A non-contiguous array can be converted to a contiguous one by the copy() method. This is useful for interfacing to C routines only, as far as I know.

>>> XXX example

typecode()

The `typecode()' method returns the typecode of the array it is applied to. While we've been talking about them as Float, Int, etc., they are represented internally as characters, so this is what you'll get:

```
>>> a = array([1,2,3])
>>> a.typecode()
'l'
>>> a = array([1], Complex)
>>> a.typecode()
'D'
```

byteswapped()

The `byteswapped` method performs a byte swapping operation on all the elements in the array.

```
>>> print a
[1 2 3]
>>> print a.byteswapped()
[16777216 33554432 50331648]
```

tostring()

The `tostring` method returns a string representation of the data portion of the array it is applied to.

```
>>> a = arange(65,100)
```

```
>>> print a.tostring()
A   B   C   D   E   F   G   H   I   J   K   L   M   N   O   P   Q   R   S   T
U   V   W   X   Y   Z   [   \   ]   ^   _   `   a   b   c
```

tolist()

Calling an array's tolist() method returns a hierarchical python list version of the same array:

```
>>> print a
[[65 66 67 68 69 70 71]
 [72 73 74 75 76 77 78]
 [79 80 81 82 83 84 85]
 [86 87 88 89 90 91 92]
 [93 94 95 96 97 98 99]]
>>> print a.tolist()
[[65, 66, 67, 68, 69, 70, 71], [72, 73, 74, 75, 76, 77, 78], [79, 80,
81, 82, 83, 84, 85], [86, 87, 88, 89, 90, 91, 92], [93, 94, 95, 96, 97,
98, 99]]
```

10. Array Attributes

We've already seen a very useful attribute of arrays, the shape attribute. There are three more, flat, real and imaginary.

flat

Accessing the `flat` attribute of an array returns the flattened, or `ravel()`'ed version of that array, without having to do a function call. The returner array has the same number of elements as the input array, but is of rank-1. One cannot set the flat attribute of an array, but one can use the indexing and slicing notations to modify the contents of the array:

```
>>> print a
[[0 1 2]
 [3 4 5]
 [6 7 8]]
>>> print a.flat
[0 1 2 3 4 5 6 7 8]
>>> a.flat = arange(9,18)
Traceback (innermost last):
  File "<stdin>", line 1, in ?
AttributeError: Attribute does not exist or cannot be set
>>> a.flat[4] = 100
>>> print a
[[  0   1   2]
 [  3 100   5]
 [  6   7   8]]
>>> a.flat[:] = arange(9, 18)
>>> print a
[[ 9 10 11]
 [12 13 14]
 [15 16 17]]
```

real and *imaginary*

These attributes exist only for complex arrays. They return respectively arrays filled with the real and imaginary parts of their elements. `.imag` is a synonym for `.imaginary`. The arrays returned are not contiguous (except for arrays of length 1, which are always contiguous.). `.real`, `.imag` and `.imaginary` are modifiable:

```
>>> print x
[ 0.       +1.j         0.84147098+0.54030231j  0.90929743-0.41614684j]
>>> print x.real
[ 0.          0.84147098  0.90929743]
>>> print x.imag
[ 1.          0.54030231 -0.41614684]
>>> x.imag = arange(3)
>>> print x
[ 0.        +0.j  0.84147098+1.j  0.90929743+2.j]
>>> x = reshape(arange(10), (2,5)) + 0j# make complex array
>>> print x
[[ 0.+0.j  1.+0.j  2.+0.j  3.+0.j  4.+0.j]
```

```
 [ 5.+0.j  6.+0.j  7.+0.j  8.+0.j  9.+0.j]]
>>> print x.real
[[ 0.  1.  2.  3.  4.]
 [ 5.  6.  7.  8.  9.]]
>>> print x.typecode(), x.real.typecode()
D d
>>> print x.itemsize(), x.imag.itemsize()
16 8
```

11. Special Topics

This chapter holds miscellaneous information which did not neatly fit in any of the other chapters.

Code Organization

Numeric.py and friends

`Numeric.py` is the most commonly used interface to the Numeric extensions. It is a Python module which imports all of the exported functions and attributes from the `multiarray` module, and then defines some utility functions. As some of the functions defined in `Numeric.py` could someday be moved into a supporting C module, the utility functions and the `multiarray` object are documented together, in this section. The `multiarray` objects are the core of Numeric Python – they are extension types written in C which are designed to provide both space- and time-efficiency when manipulating large arrays of homogeneous data types, with special emphasis to numeric data types.

UserArray.py

In the tradition of `UserList.py` and `UserDict.py`, the `UserArray.py` module defines a class whose instances act in many ways like array objects.

Matrix.py

The `Matrix.py` python module defines a class `Matrix` which is a subclass of `UserArray`. The only differences between `Matrix` instances and `UserArray` instances is that the `*` operator on `Matrix` performs a matrix multiplication, as opposed to element-wise multiplication, and that the power operator `**` is disallowed for `Matrix` instances.

Precision.py

The Precision.py module contains the code which is used to determine the mapping between typecode names and values, by building small arrays and looking at the number of bytes they use per element.

ArrayPrinter.py

The ArrayPrinter.py module defines the functions used for default printing of arrays. See the section on Textual Representations of arrays on page 63,

Mlab.py

The Mlab.py module provides some functions which are compatible with the functions of the same name in the MATLAB programming language. These are:

bartlett(M)

returns the M-point Bartlett window.

blackman(M)

returns the M-point Blackman window.

corrcoef(x, y=None)

The correlation coefficient

cov(m,y=None)

returns the covariance

cumprod(m)

returns the cumulative product of the elments along the first dimension of m.

cumsum(m)

returns the cumulative sum of the elements along the first dimension of m.

diag(v, k=0)

returns the k-th diagonal if v is a matrix or returns a matrix with v as the k-th diagonal if v is a vector.

diff(x, n=1)

calculates the first-order, discrete difference approximation to the derivative.

eig(m)

returns the the eigenvalues of m in x and the corresponding eigenvectors in the rows of v.

eye(N, M=N, k=0, typecode=None)

returns a N-by-M matrix where the k-th diagonal is all ones, and everything else is zeros.

fliplr(m)

returns a 2-D matrix m with the rows preserved and columns flipped in the left/right direction. Only works with 2-D arrays.

flipud(m)

returns a 2-D matrix with the columns preserved and rows flipped in the up/down direction. Only works with 2-D arrays.

hamming(M)

returns the M-point Hamming window.

hanning(M)

returns the M-point Hanning window.

kaiser(M, beta)

returns a Kaiser window of length M with shape parameter beta. It depends on the cephes module for the modified bessel function i0.

max(m)

returns the maximum along the first dimension of m.

mean(m)

returns the mean along the first dimension of m. Note: if m is an integer array, integer division will occur.

median(m)

returns a mean of m along the first dimension of m.

min(m)

returns the minimum along the first dimension of m.

msort(m)

returns a sort along the first dimension of m as in MATLAB.

prod(m)

returns the product of the elements along the first dimension of m.

ptp(m)

returns the maximum - minimum along the first dimension of m.

rand(d1, ..., dn)

returns a matrix of the given dimensions which is initialized to random numbers from a uniform distribution in the range [0,1).

rot90(m,k=1)

returns the matrix found by rotating m by k*90 degrees in the counterclockwise direction.

sinc(x)

returns sin(pi*x)/(pi*x) at all points of array x.

squeeze(a)

removes any ones from the shape of a

std(m)

returns the standard deviation along the first dimension of m. The result is unbiased meaning division by len(m)-1.

sum(m)

returns the sum of the elements along the first dimension of m.

svd(m)

return the singular value decomposition of m [u,x,v]

trapz(y,x=None)

integrates y = f(x) using the trapezoidal rule.

tri(N, M=N, k=0, typecode=None)

returns a N-by-M matrix where all the diagonals starting from lower left corner up to the k-th are all ones.

tril(m,k=0)

returns the elements on and below the k-th diagonal of m. k=0 is the main diagonal, k > 0 is above and k < 0 is below the main diagonal.

triu(m,k=0)

returns the elements on and above the k-th diagonal of m. k=0 is the main diagonal, k > 0 is above and k < 0 is below the main diagonal.

The multiarray object

The array objects which Numeric Python manipulates is actually a multiarray object, given this name to distinguish it from the one-dimensional array object defined in the standard array module. From here on, however, the terms array and multiarray will be used interchangeably to refer to the new object type. multiarray objects are homogeneous multidimensional sequences. Starting from the back, they are sequences. This means that they are container (compound) objects, which contain references to other objects. They are multidimensional, meaning that unlike standard Python sequences which define only a single dimension along which one can iterate through the contents, multiarray objects can have up to 40 dimensions.[1] Finally, they are homogeneous. This means that every object in a multiarray must be of the same type. This is done for efficiency reasons -- storing the type of the contained objects once in the array means that the process of finding the type-specific operation to operate on each element in the array needs to be done only once per array, as opposed to once per element. Furthemore, as the main purpose of these arrays is to process numbers, the numbers can be stored directly, and not as full-fledged Python objects (PyObject *), thus yielding memory savings. It is however possible to make arrays of Python objects, which relinquish both the space and time efficiencies but allow heterogeneous contents (as we shall see, these arrays are still homogeneous from the Numeric perspective, they are just arrays of Python object references).

Typecodes

The kind of number stored in an array is described by its typecode. This code is stored internally as a single-character Python string, but more descriptive names corresponding to the typecodes are made available to the Python programmer in the Precision.py module. The typecodes are defined as follows:

Table 3: Typecode Listing

Variable defined in `Typecode` module	Typecode character	Description
`Char`	`'c'`	Single-character strings
`PyObject`	`'O'`	Reference to Python object
`UnsignedInt8`	`'b'`	Unsigned integer using a single byte.
`Int`	`'l'`	Python standard integers (i.e. C long integers)
`Float`	`'d'`	Python standard floating point numbers (i.e. C double-precision floats)
n/a	`'f'`	Single-precision floating point numbers
`Complex`	`'D'`	Complex numbers consisting of two double-precision floats
n/a	`'F'`	Complex numbers consisting of two single-precision floats

1. This limit is modifiable in the source code if higher dimensionality is needed.

Table 3: Typecode Listing

Variable defined in `Typecode` module	Typecode character	Description
`Int0, Int8, Int16, Int32, Int64, Int128`	n/a	These correspond to machine-dependent typecodes: `Int0` returns the typecode corresponding to the smallest available integer, `Int8` that corresponding to the smallest available integer with at least 8 bits, `Int16` that with at least 16 bits, etc. If a typecode is not available (e.g. `Int64` on a 32-bit machine), the variable is not defined.
`Float0, Float8, Float16, Float32, Float64, Float128`	n/a	Same as `Int0`, `Int8` etc. except for floating point numbers.
`Complex0, Complex8, Complex16, Complex32, Complex64, Complex128`	n/a	Same as `Float0`, etc., except that the number of bits refers to the precision of each of the two (real and imaginary) parts of the complex number.

Note on number fomat: the binary format used by Python is that of the underlying C library. [notes about IEEE formats, etc?]

Indexing in and out, slicing

Indexing arrays works like indexing of other Python sequences, but supports some extensions which are as of yet not implemented for other sequence types[1]. The standard [start:stop] notation is supported, with start defaulting to 0 (the first index position) and stop defaulting to the length of the sequence, as for lists and tuples. In addition, there is an optional stride argument, which specifies the stride size between successive indices in the slice. It is expressed by a integer following a second : immediately after the usual start:stop slice. Thus `[0:11:2]` will slice the array at indices 0, 2, 4, .. 10. The start and stop indices are optional, but the first : must be specified for the stride interpretation to occur. Therefore, `[::2]` means slice from beginning to end, with a stride of 2 (i.e. skip an index for each stride). If the start index is omitted and the stride is negative, the indexing starts from the end of the sequence and works towards the beginning of the sequence. If the stop index is omitted and the stride is negative, the indexing stops at the beginning of the sequence.

```
>>> print x
[ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19]
>>> print x[10]
10
>>> print x[:10]
[0 1 2 3 4 5 6 7 8 9]
>>> print x[5:15:3]
[ 5  8 11 14]
>>> print x[:10:2]
[0 2 4 6 8]
>>> print x[10::-2]
[10  8  6  4  2  0]
>>> print x[::-1]
[19 18 17 16 15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0]
```

1. The Python syntax can allow other Python datatypes to use both the stride notation and multidimensional indexing, and it is relatively simple to write Python classes which support these operations. See the Python Reference manual for details.

It is important to note that the out-of-bounds conditions follow the same rules as standard Python indexing, so that slices out of bounds are trimmed to the sequence boundaries, but element indexing with out-of-bound indices yields an IndexError:

```
>>> print x[:100]
[ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19]
>>> print x[-200:4]
[0 1 2 3]
>>> x[100]
Traceback (innermost last):
  File "<stdin>", line 1, in ?
IndexError: index out of bounds
```

The second difference between array indexing and other sequences is that arrays provide multidimensional indexing. An array of rank N can be indexed with up to N indices or slices (or combinations thereof. Indices should be integers (with negative integers indicating offsets from the end of the dimension, as for other Python sequences), and slices can have, as explained above, one or two :'s separating integer arguments. These indices and slies must be separated by commas, and correspond to sequential dimensions starting from the leftmost (first) index on. Thus `a[3]` means index 3 along dimension 0. `a[3,:,-4]` means the slice of a along three dimensions: index 3 along the first dimension, the entire range of indices along the second dimension, and the 4th from the end index along the third dimension. If the array being indexed has more dimensions than are specified in the multidimensional slice, those dimensions are assumed to be sliced from beginning to end. Thus, if `a` is a rank 3 array,

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

Ellipses

A special slice element called Ellipses (and written `...`) is used to refer to a variable number of slices from beginning to end along the current dimension. It is a shorthand for a set of such slices, specifically the number of dimensions of the array being indexed minus those which are already specified. Only the first (leftmost) Ellipses in an multidimensional slice is expanded, while the others are single dimensional slices from beginning to end.

Thus, if `a` is a rank-6 array,

`a[3,:,:,:,-1,:] == a[3,...,-1,:] == a[3,...,-1,...]`.

NewAxis

There is another special symbol which can be used inside indexing operations to create new dimensions in the returned array. The reference NewAxis, used as one of the comma-separated slice elements, does not change the selection of the subset of the array being indexed, but changes the shape of the array returned by the indexing operation, so that an additional dimension (of length 1) is created, at the dimension position corresponding to the location of NewAxis within the indexing sequence. Thus, `a[:,3,NewAxis,-3]` will perform the indexing of a corresponding to the slice `[a:,3,-3]`, but will also modify the shape of a so that the new shape of `a` is `(a.shape[0], a.shape[1], 1, a.shape[2])`. This operation is especially useful in conjunction with the broadcasting feature described next, as it replaces a lengthy but common operation with a simple notation (in the example above, the same effect can be had with

`reshape(a[:,3,-1], (a.shape[0], a.shape[1], 1, a.shape[2]))`.

Set-indexing and Broadcasting

The indexing rules described so far specify exactly the behavior of get-indexing. For set-indexing, the rules are exactly the same, and describe the slice of the array on the left hand side of the assignment operator which is the target of the assignment. The only point left to mention is the process of assigning from the source (on the right hand side of the assignment) to the target (on the left hand side).

If both source and target have the same shape, then the assignment is done element by element. The typecode of the target specifies the casting which can be applied in the case of a typecode mismatch between source and target. If the typecode of the source is “lower” than that of the target, then an ’up-cast’ is performed and no loss in precision results. If the typecode of the source is “higher” than that of the target, then a downcast is performed, which may lose precision (as discussed in the description of the array call, these casts are truncating casts, not rounding casts). Complex numbers cannot be cast to non-complex numbers.

If the source and the target have different shapes, Numeric Python attempts to broadcast the contents of the source over the range of the target. This broadcasting occurs for all dimensions where the source has dimension 1 or 0 (i.e., is absent). If there exists a dimension for which the two arrays have differing lengths, and the length of that dimension in the source is not 1, then the assignment fails and an exception (ValueError) is raised, notifying the user that the arrays are not aligned.

Axis specifications

In many of the functions defined in this document, indices are used to refer to axes. The numbering scheme is the same as that used by indexing in Python: the first (leftmost) axis is axis 0, the second axis is axis 1, etc. Axis -1 refers to the last axis, -2 refers to the next-to-last axis, etc.

Textual representations of arrays

The algorithm used to display arrays as text strings is defined in the file ArrayPrinter.py, which defines a function array2string (imported into Numeric’s namespace) which offers considerable control over how arrays are output. The range of options to the array2string function will be described first, followed by a description of which options are used by default by `str` and `repr`.

array2string(a, max_line_width = None, precision = None,
suppress_small = None, separator=' ', array_output=0):

The `array2string` function takes an array and returns a textual representation of it. Each dimension is indicated by a pair of matching square brackets (`[]`), within which each subset of the array is output. The orientation of the dimensions is as follows: the last (rightmost) dimension is always horizontal, so that the frequent rank-1 arrays use a minimum of screen real-estate. The next-to-last dimension is displayed vertically if present, and any earlier dimension is displayed with additional bracket divisions. For example:

```
>>> a = arange(24)
>>> print array2string(a)
[ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23]
>>> a.shape = (2,10)
>>> print array2string(a)
[[ 0  1  2  3  4  5  6  7  8  9 10 11]
 [12 13 14 15 16 17 18 19 20 21 22 23]]
>>> a.shape = (2,3,4)
>>> print array2string(a)
[[[ 0  1  2  3]
  [ 4  5  6  7]
  [ 8  9 10 11]]
 [[12 13 14 15]
  [16 17 18 19]
  [20 21 22 23]]]
```

The `max_line_width` argument specifies the maximum number of characters which the array2string routine uses in a single line. If it is set to `None`, then the value of the `sys.output_line_width` attribute is looked up. If it exists, it is used. If not, the default of 77 characters is used.

```
>>> print array2string(x)
[ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25
```

```
      26 27 28 29]
>>> sys.output_line_width = 30
>>> print array2string(x)
[ 0  1  2  3  4  5  6  7  8  9
      10 11 12 13 14 15 16 17
      18 19 20 21 22 23 24 25
      26 27 28 29]
```

The `precision` argument specifies the number of digits after the decimal point which are used. If a value of `None` is used, the value of the `sys.float_output_precision` is looked up. If it exists, it is used. If not, the default of 8 digits is used.

```
>>> x = array((10.11111111111123123111, pi))
>>> print array2string(x)
[ 10.11111111   3.14159265]
>>> print array2string(x, precision=3)
[ 10.111   3.142]
>>> sys.float_output_precision = 2
>>> print array2string(x)
[ 10.11   3.14]
```

The `suppress_small` argument specifies whether small values should be suppressed (and output as 0). If a value of `None` is used, the value of the `sys.float_output_suppress_small` is looked up. If it exists, it is used (all that matters is whether it evaluates to true or false). If not, the default of 0 (false) is used. This variable also interacts with the precision parameters, as it can be used to suppress the use of exponential notation.

```
>>> print x
[ 1.00000000e-005  3.14159265e+000]
>>> print array2string(x)
[ 1.00000000e-005  3.14159265e+000]
>>> print array2string(x, suppress_small=1)
[ 0.00001     3.14159265]
>>> print array2string(x, precision=3)
[ 1.000e-005  3.142e+000]
>>> print array2string(x, precision=3, suppress_small=1)
[ 0.     3.142]
```

The `separator` argument is used to specify what character string should be placed between two numbers which do not straddle a dimension. The default is a single space.

```
>>> print array2string(x)
[   0  100  200  300  400  500  600  700  800  900 100]
>>> print array2string(x, separator = ', ')
[   0,  100,  200,  300,  400,  500,  600,  700,  800,  900, 100]
```

Finally, the last attribute, array_output, specifies whether to prepend the string "array(" and append either the string ")" or ", *'X'*)" where *X* is a typecode for non-default typecodes (in other words, the typecode will only be displayed if it is not that corresponding to Float, Complex or Int, which are the standard typecodes associated with floating point numbers, complex numbers and integers respectively). The array() is so that an eval of the returned string will return an array object (provided a comma separator is also used).

```
>>> array2string(arange(3))
[0 1 2]
>>> eval(array2string(arange(3), array_output=1))
Traceback (innermost last):
  File "<stdin>", line 1, in ?
  File "<string>", line 1
```

```
    array([0 1 2])
             ^
SyntaxError: invalid syntax
>>> type(eval(array2string(arange(3), array_output=1, separator=',')))
<type 'array'>
>>> array2string(arange(3), array_output=1)
'array([0, 1, 2])'
>>> array2string(zeros((3,), 'i') + arange(3), array_output=1)
"array([0, 1, 2],'i')"
```

The `str` and `repr` operations on arrays call `array2string` with the `max_line_width`, `precision` and `suppress_small` all set to None, meaning that the defaults are used, but that modifying the attributes in the `sys` module will affect array printing. str uses the default separator and does not use the array() text, while repr uses a comma as a separator and does use the array(...) text.

```
>>> x = arange(3)
>>> print x
[0 1 2]
>>> str(x)
'[0 1 2]'
>>> repr(x)
'array([0, 1, 2])'              # note the array(...) and ,'s
>>> x = arange(0,.01,.001)
>>> print x
[ 0.     0.001  0.002  0.003  0.004  0.005  0.006  0.007  0.008  0.009]
>>> import sys
>>> sys.float_output_precision = 2
>>> print x
[ 0.    0.    0.    0.    0.    0.01  0.01  0.01  0.01  0.01]
```

Comparisons

Currently, comparisons of multiarray objects results in exceptions, since reasonable results (arrays of booleans) are not doable without non-trivial changes to the Python core. These changes are planned for Python 1.6, at which point array object comparisons will be updated.

```
>>> print x, y
[0 1 2] [3 4 5]
>>> print x < y
Traceback (innermost last):
  File "<stdin>", line 1, in ?
TypeError: Comparison of multiarray objects is not implemented.
```

Pickling and Unpickling -- storing arrays on disk

HowTo

byte-order independence

Dealing with floating point exceptions

Dealing with floating point exceptions

fpectl, NaNs, etc.

12. Writing a C extension to NumPy

Introduction

There are two applications that require using the NumPy array type in C extension modules:

- Access to numerical libraries: Extension modules can be used to make numerical libraries written in C (or languages linkable to C, such as Fortran) accessible to Python programs. The NumPy array type has the advantage of using the same data layout as arrays in C and Fortran.
- Mixed-language numerical code: In most numerical applications, only a small part of the total code is CPU time intensive. Only this part should thus be written in C, the rest can be written in Python. NumPy arrays are important for the interface between these two parts, because they provide equally simple access to their contents from Python and from C.

This document is a tutorial for using NumPy arrays in C extensions.

Preparing an extension module for NumPy arrays

To make NumPy arrays available to an extension module, it must include the header file `arrayobject.h`, after the header file Python.h that is obligatory for all extension modules. The file `arrayobject.h` comes with the NumPy distribution; depending on where it was installed on your system you might have to tell your compiler how to find it. In addition to including `arrayobject.h`, the extension must call `import_array()` in its initialization function, after the call to `Py_InitModule()`. This call makes sure that the module which implements the array type has been imported, and initializes a pointer array through which the NumPy functions are called. If you forget this call, your extension module will crash on the first call to a NumPy function! If you will be manipulating ufunc objects, you should also include the file `ufuncobject.h`, also available as part of the NumPy distribution in the `Include` directory.

All of the rules related to writing extension modules for Python apply. The reader unfamiliar with these rules is encouraged to read the standard text on the topic, “Extending and Embedding the Python Interpreter,” available as part of the standard Python documentation distribution.

Accessing NumPy arrays from C

Types and Internal Structure

NumPy arrays are defined by the structure `PyArrayObject`, which is an extension of the structure `PyObject`. Pointers to `PyArrayObject` can thus safely be cast to `PyObject` pointers, whereas the inverse is safe only if the object is known to be an array. The type structure corresponding to array objects is `PyArray_Type`. The structure `PyArrayObject` has four elements that are needed in order to access the array's data from C code:

`int nd`

The number of dimensions in the array.

`int *dimensions`

A pointer to an array of `nd` integers, describing the number of elements along each dimension. The sizes are in the conventional order, so that for any array `a`, `a.shape==(dimensions[0], dimensions[1], ..., dimensions[nd])`.

`int *strides`

A pointer to an array of `nd` integers, describing the address offset between two successive data elements along each dimension. Note that strides can also be negative! Each number gives the number of bytes to add to a pointer to get to the next element in that dimension. For example, if `myptr` currently points to element of a rank-5 array at indices `1,0,5,3,2` and you want it to point to element `1,0,5,4,2` then you should add `strides[3]` to the pointer: `myptr += strides[3]`. This works even if (and is especially useful when) the array is not contiguous in memory.

`char *data`

A pointer to the first data element of the array.

The address of a data element can be calculated from its indices and the data and strides pointers. For example, element `[i, j]` of a two-dimensional array has the address `data + i*array->strides[0] + j*array->strides[1]`. Note that the stride offsets are in bytes, not in storage units of the array elements. Therefore address calculations must be made in bytes as well, starting from the data pointer, which is always a char pointer. To access the element, the result of the address calculation must be cast to a pointer of the required type. The advantage of this arrangement is that purely structural array operations (indexing, extraction of subarrays, etc.) do not have to know the type of the array elements.

Element data types

The type of the array elements is indicated by a type number, whose possible values are defined as constants in `arrayobject.h`, as given in Table 3.

Table 4: C constants corresponding to storage types

Constant	element data type
`PyArray_CHAR`	`char`
`PyArray_UBYTE`	`unsigned char`
`PyArray_SBYTE`	`signed char`
`PyArray_SHORT`	`short`
`PyArray_INT`	`int`
`PyArray_LONG`	`long`
`PyArray_FLOAT`	`float`
`PyArray_DOUBLE`	`double`
`PyArray_CFLOAT`	`float[2]`
`PyArray_CDOUBLE`	`double[2]`
`PyArray_OBJECT`	`PyObject *`

The type number is stored in `array->descr->type_num`. Note that the names of the element type constants refer to the C data types, not the Python data types. A Python `int` is equivalent to a C `long`, and a Python `float` corresponds to a C `double`. Many of the element types listed above do not have corresponding Python scalar types (e.g. `PyArray_INT`).

Contiguous arrays

An important special case of a NumPy array is the contiguous array. This is an array whose elements occupy a single contiguous block of memory and have the same order as a standard C array. In a contiguous array, the value of `array->strides[i]` is equal to the size of a single array element times the product of `array->dimensions[j]` for `j` up to `i-1`. Arrays that are created from scratch are always contiguous; non-contiguous arrays are the result of indexing and other structural array operations. The main advantage of contiguous arrays is easier handling in C; the pointer `array->data` is cast to the required type and then used like a C array, without any reference to the stride values. This is particularly important when interfacing to existing libraries in C or Fortran, which typically require this standard data layout. A function that requires input arrays to be contiguous must call the conversion function `PyArray_ContiguousFromObject()`, described in the section “Accepting input data from any sequence type".

Zero-dimensional arrays

NumPy permits the creation and use of zero-dimensional arrays, which can be useful to treat scalars and higher-dimensional arrays in the same way. However, library routines for general use should not return zero-demensional arrays, because most Python code is not prepared to handle them. Moreover, zero-dimensional arrays can create confusion because they behave like ordinary Python scalars in many circumstances but are of a different type. A comparison between a Python scalar and a zero-dimensional array will always fail, for example, even if the values are the same. NumPy provides a conversion function from zero-dimensional arrays to Python scalars, which is described in the section “Returning arrays from C functions".

A simple example

The following function calculates the sum of the diagonal elements of a two-dimensional array, verifying that the array is in fact two-dimensional and of type `PyArray_DOUBLE`.

```
static PyObject *
trace(PyObject *self, PyObject *args)
{
  PyArrayObject *array;
  double sum;
  int i, n;

  if (!PyArg_ParseTuple(args, "O!",
                        &PyArray_Type, &array))
    return NULL;
  if (array->nd != 2 || array->descr->type_num != PyArray_DOUBLE) {
    PyErr_SetString(PyExc_ValueError,
                    "array must be two-dimensional and of type float");
    return NULL;
  }

  n = array->dimensions[0];
  if (n > array->dimensions[1])
    n = array->dimensions[1];
  sum = 0.;
  for (i = 0; i < n; i++)
    sum += *(double *)(array->data + i*array->strides[0] + i*array-
>strides[1]);

  return PyFloat_FromDouble(sum);
}
```

Accepting input data from any sequence type

The example in the last section requires its input to be an array of type double. In many circumstances this is sufficient, but often, especially in the case of library routines for general use, it would be preferable to accept input data from any sequence (lists, tuples, etc.) and to convert the element type to double automatically where possible. NumPy provides a function that accepts arbitrary sequence objects as input and returns an equivalent array of specified type (this is in fact exactly what the array constructor `Numeric.array()` does in Python code):

```
PyObject *
PyArray_ContiguousFromObject(PyObject *object,
                             int type_num,
                             int min_dimensions,
                             int max_dimensions);
```

The first argument, object, is the sequence object from which the data is taken. The second argument, type_num, specifies the array element type (see the table in the section “Element data types". If you want the function to the select the ``smallest" type that is sufficient to store the data, you can pass the special value `PyArray_NOTYPE`. The remaining two arguments let you specify the number of dimensions of the resulting array, which is guaranteed to be no smaller than `min_dimensions` and no larger than `max_dimensions`, except for the case `max_dimensions == 0`, which means that no upper limit is imposed.

If the input data is not compatible with the type or dimension restrictions, an exception is raised. Since the array returned by `PyArray_ContiguousFromObject()` is guaranteed to be contiguous, this function also provides a method of converting a non-contiguous array to a contiguous one. If the input object is already a contiguous array of the specified type, it is passed on directly; there is thus no performance or memory penalty for calling the conversion function when it is not required. Using this function, the example from the last section becomes

```
static PyObject *
trace(PyObject *self, PyObject *args)
{
  PyObject *input;
  PyArrayObject *array;
  double sum;
  int i, n;

  if (!PyArg_ParseTuple(args, "O", &input))
    return NULL;
  array = (PyArrayObject *)
          PyArray_ContiguousFromObject(input, PyArray_DOUBLE, 2, 2);
  if (array == NULL)
    return NULL;

  n = array->dimensions[0];
  if (n > array->dimensions[1])
    n = array->dimensions[1];
  sum = 0.;
  for (i = 0; i < n; i++)
    sum += *(double *)(array->data + i*array->strides[0] + i*array-
>strides[1]);

  Py_DECREF(array);
  return PyFloat_FromDouble(sum);
}
```

Note that no explicit error checking is necessary in this version, and that the array reference that is returned by `PyArray_ContiguousFromObject()` must be destroyed by calling `Py_DECREF()`.

Creating NumPy arrays

NumPy arrays can be created by calling the function

```
PyObject *
PyArray_FromDims(int n_dimensions,
                 int dimensions[n_dimensions],
                 int type_num);
```

The first argument specifies the number of dimensions, the second one the length of each dimension, and the third one the element data type (see the table in the section “Element data types". The array that is returned is contiguous, but the contents of its data space are undefined. There is a second function which permits the creation of an array object that uses a given memory block for its data space:

```
PyObject *
PyArray_FromDimsAndData(int n_dimensions,
                        int dimensions[n_dimensions]
                        int item_type
                        char *data);
```

The first three arguments are the same as for `PyArray_FromDims()`. The fourth argument is a pointer to the memory block that is to be used as the array's data space. It is the caller's responsibility to ensure that this memory block is not freed before the array object is destroyed. With few exceptions (such as the creation of a temporary array object to which no reference is passed to other functions), this means that the memory block may never be freed, because the lifetime of Python objects are difficult to predict. Nevertheless, this function can be useful in special cases, for example for providing Python access to arrays in Fortran common blocks.

Returning arrays from C functions

Array objects can of course be passed out of a C function just like any other object. However, as has been mentioned before, care should be taken not to return zero-dimensional arrays unless the receiver is known to be prepared to handle them. An equivalent Python scalar object should be returned instead. To facilitate this step, NumPy provides a special function

```
PyObject *
PyArray_Return(PyArrayObject *array);
```

which returns the array unchanged if it has one or more dimensions, or the appropriate Python scalar object in case of a zero-dimensional array.

A less simple example

The function shown below performs a matrix-vector multiplication by calling the `BLAS` function `DGEMV`. It takes three arguments: a scalar prefactor, the matrix (a two-dimensional array), and the vector (a one-dimensional array). The return value is a one-dimensional array. The input values are checked for consistency. In addition to providing an illustration of the functions explained above, this example also demonstrates how a Fortran routine can be integrated into Python. Unfortunately, mixing Fortran and C code involves machine-specific peculiarities. In this example, two assumptions have been made:

- The Fortran function `DGEMV` must be called from C as `dgemv_`. Many Fortran compilers apply this rule, but the C name could also be `dgemv` or `DGEMV` (or in principle anything else; there is no fixed standard).
- Fortran `integer`s are equivalent to C `long`s, and Fortran double precision numbers are equivalent to C doubles. This works for all systems that I have personally used, but again there is no standard.

Also note that the libraries that this function must be linked to are system-dependent; on my Linux system (using `gcc`/`g77`), the libraries are `blas` and `f2c`. So here is the code:

```
static PyObject *
matrix_vector(PyObject *self, PyObject *args)
{
  PyObject *input1, *input2;
  PyArrayObject *matrix, *vector, *result;
  int dimensions[1];
  double factor[1];
  double real_zero[1] = {0.};
  long int_one[1] = {1};
  long dim0[1], dim1[1];

  extern dgemv_(char *trans, long *m, long *n,
                double *alpha, double *a, long  *lda,
                double *x, long *incx,
                double *beta, double *Y, long *incy);

  if (!PyArg_ParseTuple(args, "dOO", factor, &input1, &input2))
    return NULL;
  matrix = (PyArrayObject *)
           PyArray_ContiguousFromObject(input1, PyArray_DOUBLE, 2, 2);
  if (matrix == NULL)
    return NULL;
  vector = (PyArrayObject *)
           PyArray_ContiguousFromObject(input2, PyArray_DOUBLE, 1, 1);
  if (vector == NULL)
    return NULL;
  if (matrix->dimensions[1] != vector->dimensions[0]) {
    PyErr_SetString(PyExc_ValueError,
                    "array dimensions are not compatible");
    return NULL;
  }

  dimensions[0] = matrix->dimensions[0];
  result = (PyArrayObject *)PyArray_FromDims(1, dimensions,
PyArray_DOUBLE);
  if (result == NULL)
    return NULL;

  dim0[0] = (long)matrix->dimensions[0];
  dim1[0] = (long)matrix->dimensions[1];
  dgemv_("T", dim1, dim0, factor, (double *)matrix->data, dim1,
         (double *)vector->data, int_one,
         real_zero, (double *)result->data, int_one);

  return PyArray_Return(result);
}
```

Note that `PyArray_Return()` is not really necessary in this case, since we know that the array being returned is one-dimensional. Nevertheless, it is a good habit to always use this function; its performance cost is practically zero.

13. C API Reference

This chapter describes the API for ArrayObjects and Ufuncs.

ArrayObject C Structure and API

Structures

The PyArrayObject is, like all Python types, a kind of PyObject. Its definition is:

```
typedef struct {
  PyObject_HEAD
  char *data;
  int nd;
  int *dimensions, *strides;
  PyObject *base;
  PyArray_Descr *descr;
  int flags;
} PyArrayObject;
```

Where `PyObject_HEAD` is the standard `PyObject` header, and the other fields are:

`char *data`

A pointer to the first data element of the array.

`int nd`

The number of dimensions in the array.

`int *dimensions`

A pointer to an array of `nd` integers, describing the number of elements along each dimension. The sizes are in the conventional order, so that for any array `a`, `a.shape==(dimensions[0], dimensions[1], ..., dimensions[nd])`.

`int *strides`

A pointer to an array of `nd` integers, describing the address offset between two successive data elements along each dimension. Note that strides can also be negative! Each number gives the number of bytes to add to a pointer to get to the next element in that dimension. For example, if `myptr` currently points to an element in a rank-5 array at indices `1,0,5,3,2` and you want it to point to element `1,0,5,4,2` then you should add `strides[3]` to the pointer: `myptr += strides[3]`. This works even if (and is especially useful when) the array is not contiguous in memory.

`PyObject *base`

Used internally in arrays that are created as slices of other arrays. Since the new array shares its data area with the old one, the original array's reference count is incremented. When the subarray is garbage collected, the base array's reference count is decremented.

`PyArray_Desc *descr`

See below.

`int flags`

A bitfield indicating whether the array:

- is contiguous (rightmost bit)
- owns the dimensions (next bit to the left) (???)
- owns the strides (next bit to the left) (???)
- owns the data area

The ownership bits are used by NumPy internally to manage memory allocation and deallocation. They can be false if the array is the result of e.g. a slicing operation on an existing array.

`PyArrayDescr *descr`

a pointer to a data structure that describes the array and has some handy functions. The slots in this structure are:

`PyArray_VectorUnaryFunc *cast[]`

an array of function pointers which will cast this arraytype to each of the other data types.

`PyArray_GetItemFunc *getitem`

a pointer to a function which returns a PyObject of the appropriate type given a (char) pointer to the data to get.

`PyArray_SetItemFunc *setitem`

a pointer to a function which sets the element pointed to by the second argument to converted Python Ojbect given as the first argument.

`int type_num`

A number indicating the datatype of the array (i.e. a `PyArray_XXXX`)

`char *one`

A pointer to a representation of one for this datatype.

`char *zero`

A pointer to a representation of zero for this datatype (especially useful for PyArray_OBJECT types)

`char type`

A character representing the array's typecode (one of `'cb1silfdFDO'`).

The ArrayObject API

In the following `op` is a pointer to a `PyObject` and `arp` is a pointer to a `PyArrayObject`. Routines which return `PyObject *` return `NULL` to indicate failure (and follow the standard exception-setting mechanism). Functions followed by a dagger (†) are functions which return PyObjects whose reference count has been increased by one (new references). See the Python Extending/Embedding manual for details on reference-count management.

`int PyArray_Check(op)`

returns `1` if `op` is a `PyArrayObject` or `0` if it is not.

`int PyArray_SetNumericOps(d)`

internally used by `umath` to setup some of its functions.

`int PyArray_INCREF(op)`

Used for arrays of python objects (`PyArray_OBJECT`) to increment the reference count of every

python object in the array `op`. User code does not typically need to call this.

`int PyArray_XDECREF(op)`

Used for arrays of python objects (`PyArray_OBJECT`) to decrement the reference count of every python object in the array `op`.

`PyArrayError`

Exports the array error object. I don't know its use.

`void PyArray_SetStringFunction(op,repr)`

Sets the function for representation of all arrays to `op` which should be a callable `PyObject`. If `repr` is non-zero then the function corresponding to the `repr` string representationis set, otherwise, that for the `str` string representation is set.

`PyArray_Descr PyArray_DescrFromType(type)`

returns a `PyArray_Descr` structure for the datatype given by `type`. The input type can be either the enumerated types (`PyArray_Float`, etc.) or a character (`'cb1silfdFDO'`).

`PyObject *PyArray_Cast(arp, type)` †

returns a pointer to a `PyArrayObject` that is `arp` cast to the array type specified by `type`. It is just a wrapper around the function defined in `arp->descr->cast` that handles non-contiguous arrays and arrays of Python objects appropriately.

`int PyArray_CanCastSafely(fromtype,totype)`

returns `1` if the array with type `fromtype` can be cast to an array of type `totype` without loss of accuracy, otherwise it returns `0`. It allows conversion of `long`s to `int`s which is not safe on 64-bit machines. The inputs `fromtype` and `totype` are the enumerated array types (e.g. `PyArray_SBYTE`).

`int PyArray_ObjectType(op, min_type)`

returns the typecode to use for a call to an array creation function given an input python sequence object `op` and a minimum type value, `min_type`. It looks at the datatypes used in `op`, compares this with `min_type` and returns a consistent type value that can be used to store all of the data in `op` and satisfying at the minimum the precision of `min_type`.

`int _PyArray_multiply_list(list,n)`

is a utility routine to multiply an array of `n` integers pointed to by `list`.

`int PyArray_Size(op)`

is a useful function for returning the total number of elements in `op` if `op` is a `PyArrayObject`, `0` otherwise.

`PyObject *PyArray_FromDims(nd,dims,type)` †

returns a pointer to a newly constructed `PyArrayObject` (returned as a `PyObject`) given the number of dimensions in `nd`, an array `dims` of `nd` integers specifying the size of the array, and the enumerated type of the array in `type`.

`PyObject *PyArray_FromDimsAndData(nd,dims,type,data)` †

This function should only be used to access global data that will never be freed (like FORTRAN common blocks). It builds a `PyArrayObject` in the same way as `PyArray_FromDims` but instead of allocating new memory for the array elements it uses the bytes pointed to by `data` (a `char *`).

`PyObject *PyArray_ContiguousFromObject(op,type,min_dim,max_dim)` †

returns a contiguous array of type `type` from the (possibly nested) sequence object `op`. If `op` is a contiguous `PyArrayObject` then a reference is made; if `op` is a non-contiguous then a copy is performed to get a contiguous array; if `op` is not a `PyArrayObject` then a new `PyArrayObject` is created from the sequence object and returned. The two parameters `min_dim` and `max_dim` let you specify the expected rank of the input sequence. An error will result if the resulting `PyArrayObject` does not have rank bounded by these limits. To specify an exact rank requirement set `min_dim = max_dim`. To allow for an arbitrary number of dimensions specify `min_dim = max_dim = 0`.

`PyObject *PyArray_CopyFromObject(op,type,min_dim,max_dim)` †

returns a contiguous array similar to `PyArray_ContiguousFromObject` except that a copy of `op` is performed even if a shared array could have been used.

`PyObject *PyArray_FromObject(op,type,min_dim,max_dim)` †

returns a reference to `op` if `op` is a `PyArrayObject` and a newly constructed `PyArrayObject` if `op` is any other (nested) sequence object. You must use strides to access the elements of this possibly discontiguous array correctly.

`PyObject *PyArray_Return(apr)`

returns a pointer to `apr` with some extra code to check for errors and be sure that zero-dimensional arrays are returned as scalars.

`PyObject *PyArray_Reshape(apr,op)` †

returns a reference to `apr` with a new shape specified by `op` which must be a one dimensional sequence object. One dimension may be specified as unknown by giving a value less than zero, its value will be calculated from the size of `apr`.

`PyObject *PyArray_Copy(apr)` †

returns an element-for-element copy of `apr`

`PyObject *PyArray_Take(a,indices,axis)` †

the equivalent of `take(a, indices, axis)` which is a method defined in the Numeric module that just calls this function.

`int PyArray_As1D(*op, char **ptr, int *n, int type)`

This function replaces `op` with a pointer to a contiguous 1-D `PyArrayObject` (using `PyArray_ContiguousFromObject`) and sets as output parameters a pointer to the first byte of the array in `ptr` and the number of elements in the array in `n`. It returns `-1` on failure (`op` is not a 1-D array or sequence object that can be cast to type `type`) and `0` on success.

`int PyArray_As2D(*op, char **ptr, int *m, int *n, int type)`

This function replaces `op` with a pointer to a contiguous 2-D `PyArrayObject` (using `PyArray_ContiguousFromObject`). It returns -1 on failure (op is not a 2-D array or nested sequence object that can be cast to type type) and 0 on success. It also sets as output parameters: an array of pointers in `ptr` which can be used to access the data as a 2-D array so that ptr[i][j] is a pointer to the first byte of element [i,j] in the array; `m` and `n` are set to respectively the number of rows and columns of the array.

`int PyArray_Free(op,ptr)`

is supposed to free the allocated data structures and decrease object references when using `PyArray_As1D` and `PyArray_As2D` but there are suspicions that this code is buggy.

Notes

Number formats, overflow issues, NaN/Inf representations, fpectl module, how to deal with 'missing' values.

UfuncObject C Structure and API

C Structure

The ufuncobject is a generic function object that can be used to perform fast operations over Numeric Arrays with very useful broadcasting rules and type conversions performed automatically. The ufuncobject and its API make it easy and graceful to add arbitrary functions to Python which operate over Numeric arrays. All of the unary and binary operators currently available in the Numerical extensions (like sin, cos, +, logical_or, etc.) are implemented using this object. The hooks are all in place to make it very easy to add any function that takes one or two (double) arguments and returns a single (double) argument. It is not difficult to add support routines in order to handle arbitrary functions whose total number of input/output arguments is less than some maximum number (currently 10).

```
typedef struct {
  PyObject_HEAD
  int *ranks, *canonical_ranks;
  int nin, nout, nargs;
  int identity;
  PyUFuncGenericFunction *functions;
  void **data;
  int ntypes, nranks, attributes;
  char *name, *types;
  int check_return;
} PyUFuncObject;
```

where:

`int *ranks`

unused.

`int *canonical_ranks`

unused

`int nin`

the number of input arguments to function

`int nout`

the number of output arguments for the function

`int nargs`

the total number of arguments `= nin + nout`

`int identity`

a flag telling whether the identity for this function is `0` or `1` for use in the `reduce` method for a zero size array input.

`PyUFuncGenericFunction *functions`

an array of functions that perform the innermost looping over the input and output arrays (I think this is over a single axis). These functions call the underlying math function with the data from the input arguments along this axis and return the outputs of the function into the correct place in the output arrayobject (with appropriate typecasting). These functions are called by the general looping code. There is one function for each of the supported datatypes. Function pointers to do this looping for

types `'f'`, `'d'`, `'F'`, and `'D'`, are provided in the C-API for functions that take one or two arguments and return one argument. Each `PyUFuncGenericFunction` returns `void` and has the following argument list (in order):

`args`

an array of pointers to the data for each of the input and output arguments with input arguments first and output arguments immediately following. Each element of `args` is a `char *` to the first byte in the corresponding input or output array.

`dimensions`

a pointer to a single `int` giving the size of the axis being looped over.

`steps`

an array of `int`s giving the number of bytes to skip to go to the next element of the array for this loop. There is an entry in the array for each of the input and output arguments, with input arguments first and output arguments immediately following.

`func`

a pointer to the underlying math function to be called at each point in this inner loop. This is a `void *` and must be recast to the required type before actually calling the function e.g. to a pointer to a function that takes two `double`s and returns a `double`). If you need to write your own `PyUFuncGenericFunction`, it is most readable to also have a `typedef` statement that defines your specific underlying function type so the function pointer cast is somewhat readable.

`void **data`

a pointer to an array of functions (each cast to `void *`) that compute the actual mathematical function for each set of inputs and outputs. There should be a function in the array for each supported data type. This function will be called from the `PyUFuncGenericFunction` for the corresponding type.

`int ntypes`

the number of datatypes supported by this function. For datatypes that are not directly supported, a coercion will be performed if possible safely, otherwise an error will be reported.

`int nranks`

unused.

`int attributes`

unused.

`char *name`

the name of this function (not the same as the dictionary label for this function object, but it is usually set to the same string). It is printed when `__repr__` is called for this object, defaults to `"?"` if set to `NULL`.

`char *types`

an array of supported types for this function object. I'm not sure why but each supported datatype (`PyArray_FLOAT`, etc.) is entered as many times as there are arguments for this function. (`nargs`)

`int check_return`

Usually best to set to 1. If this is non-zero then returned matrices will be cleaned up so that rank-0 arrays will be returned as python scalars. Also, if non-zero, then any math error that sets the `errno` global variable will cause an appropriate Python exception to be raised.

UfuncObject C API

There are currently 15 pointers in the C-API array for the ufuncobject which is loaded by `import_ufunc()`. The macros implemented by this API, available by including the file `ufuncobject.h`,' are given below. The only function normally called by user code is the ufuncobject creation function `PyUFunc_FromFuncAndData`. Some of the other functions can be used as elements of an array to be passed to this creation function.

`int PyUFunc_Check(op)`

returns `1` if op is a ufunc object otherwise returns `0`.

`PyObject *PyUFunc_FromFuncAndData(functions, data, types, ntypes, nin, nout, identity, name, check_return)`

returns the ufunc object given its parameters. This is the most important function call. It requires defining three arrays to be passed as parameters: `functions`, `data`, and `types`. The arguments to be passed are:

`functions`

an array of functions of type `PyUFuncGenericFunction`, there should be one function for each supported datatype. The functions should be in order so that datatypes listed toward the beginning of the array could be cast as datatypes listed toward the end.

`data`

an array of pointers to void* the same size as the functions array and in the same datatype order. Each element of this array is the actual underlying math function (recast to a `void *`) that will be called from one of the `PyUFuncGenericFunctions`. It will operate on each element of the input NumPy `arrayobject`(s) and return its element-by-element result in the output NumPy arrayobject(s). There is one function call for each datatype supported, (though functions can be repeated if you handle the typecasting appropriately with the `PyUFuncGenericFunction`).

`types`

an array of `PyArray_Type`s. The size of this array should be (`nin+nout`) times the size of one of the previous two arrays. There should be `nin+nout` copies of `PyArray_XXXXX` for each datatype explicitly supported. (Remember datatypes not explicitly supported will still be accepted as input arguments to the ufunc if they can be cast safely to a supported type.)

`ntypes`

the number of supported types for this ufunc.

`nin`

the number of input arguments

`nout`

the number of output arguments

`identity`

`PyUFunc_One`, `PyUFunc_Zero`, or `PyUFunc_None`, depending on the desired value for the identity. This is only relevant for functions that take two input arguments and return one output argument. If not relevant use `PyUFunc_None`.

`name`

the name of this `ufuncobject` for use in the `__repr__` method.

`check_return`

the desired value for check_return for this ufuncobject.

`int PyUFunc_GenericFunction(self,args,mps)`

allows calling the ufunc from user C routine. It returns 0 on success and -1 on any failures. This is the core of what happens when a ufunc is called from Python. Its arguments are:

`self`

the ufunc object to be called. INPUT

`args`

a Python tuple object containing the input arguments to the ufunc (should be Python sequence objects). INPUT

`mps`

an array of pointers to PyArrayObjects for the input and output arguments to this function. The input NumPy arrays are elements `mps[0]...mps[self->nin-1]`. The output NumPy arrays are elements `mps[self->nin]...mps[self->nargs-1]`. OUTPUT

The following are all functions of type `PyUFuncGenericFunction` and are suitable for use in the `functions` argument passed to `PyUFunc_FromFuncAndData`:

`PyUFunc_f_f_As_d_d`

for a unary function that takes a `double` input and returns a `double` output as a ufunc that takes `PyArray_FLOAT` input and returns `PyArray_FLOAT` output.

`PyUFunc_d_d`

for a using a unary function that takes a `double` input and returns a `double` output as a ufunc that takes `PyArray_DOUBLE` input and returns `PyArray_DOUBLE` output.

`PyUFunc_F_F_As_D_D`

for a unary function that takes a `Py_complex` input and returns a `Py_complex` output as a ufunc that takes `PyArray_CFLOAT` input and returns `PyArray_CFLOAT` output.

`PyUFunc_D_D`

for a unary function that takes a `Py_complex` input and returns a `Py_complex` output as a ufunc that takes `PyArray_CFLOAT` input and returns `PyArray_CFLOAT` output.

`PyUFunc_O_O`

for a unary function that takes a `Py_Object *` input and returns a `Py_Object *` output as a ufunc that takes `PyArray_OBJECT` input and returns `PyArray_OBJECT` output

`PyUFunc_ff_f_As_dd_d`

for a binary function that takes two `double` inputs and returns one `double` output as a ufunc that takes `PyArray_FLOAT` input and returns `PyArray_FLOAT` output.

`PyUFunc_dd_d`

for a binary function that takes two `double` inputs and returns one `double` output as a ufunc that takes `PyArray_DOUBLE` input and returns `PyArray_DOUBLE` output.

`PyUFunc_FF_F_As_DD_D`

for a binary function that takes two `Py_complex` inputs and returns a `Py_complex` output as a ufunc that takes `PyArray_CFLOAT` input and returns `PyArray_CFLOAT` output.

`PyUFunc_DD_D`

for a binary function that takes two `Py_complex` inputs and returns a `Py_complex` output as a ufunc that takes `PyArray_CFLOAT` input and returns `PyArray_CFLOAT` output

`PyUFunc_OO_O`

for a unary function that takes two `Py_Object *` input and returns a `Py_Object *` output as a ufunc that takes `PyArray_OBJECT` input and returns `PyArray_OBJECT` output

`PyUFunc_O_O_method`

for a unary function that takes a `Py_Object *` input and returns a `Py_Object *` output and is pointed to by a Python method as a ufunc that takes PyArray_OBJECT input and returns PyArray_OBJECT output

`PyArrayMap`

an exported API that was apparently considered but never implemented probably because the functionality is already available with Python's `map` function.

14. FFT Reference

The FFT.py module provides a simple interface to the FFTPACK FORTRAN library, which is a powerful standard library for doing fast Fourier transforms of real and complex data sets, or the C fftpack library, which is algorithmically based on FFTPACK and provides a compatible interface. On some platforms, optimized version of one of these libraries may be available, and can be used to provide optimal performance (see “Compilation Notes” on page 83).

Python Interface

The Python user imports the FFT module, which provides a set of utility functions which provide access to the most commonly used FFT routines, and allows the specification of which axes (dimensions) of the input arrays are to be used for the FFT’s. These routines are:

fft(data, n=None, axis=-1)

Performs a n-point discrete Fourier transform of the array data. `n` defaults to the size of data. It is most efficient for n a power of two. If n is larger than `data`, then `data` will be zero-padded to make up the difference. If n is smaller than data, then `data` will be aliased to reduce its size. This also stores a cache of working memory for different sizes of fft's, so you could theoretically run into memory problems if you call this too many times with too many different n's.

The FFT is performed along the axis indicated by the `axis` argument, which defaults to be the last dimension of `data`.

The format of the returned array is a complex array of the same shape as `data`, where the first element in the result array contains the DC (steady-state) value of the FFT, and where each successive ...XXX

Example of use:

```
>>> print fft(array((1,0,1,0,1,0,1,0))+ 10).real
[ 84.   0.   0.   0.   4.   0.   0.   0.]
>>> print fft(array((0,1,0,1,0,1,0,1))+ 10).real
[ 84.   0.   0.   0.  -4.   0.   0.   0.]
>>> print fft(array((0,1,0,0,0,1,0,0))+ 10).real
[ 82.   0.   0.   0.  -2.   0.   0.   0.]
```

inverse_fft(data, n=None, axis=-1)

Will return the `n` point inverse discrete Fourier transform of `data`. `n` defaults to the length of `data`. This is most efficient for `n` a power of two. If `n` is larger than `data`, then `data` will be zero-padded to make up the difference. If `n` is smaller than `data`, then `data` will be aliased to reduce its size. This also stores a cache of working memory for different sizes of FFT’s, so you could theoretically run into memory problems if you call this too many times with too many different `n`'s.

real_fft(data, n=None, axis=-1)

Will return the `n` point discrete Fourier transform of the real valued array `data`. `n` defaults to the length of `data`. This is most efficient for `n` a power of two. The returned array will be one half of the symmetric complex transform of the real array.

```
>>> x = cos(arange(30.0)/30.0*2*pi)
>>> print real_fft(x)
[ -1.        +0.j          13.69406641+2.91076367j
       -0.91354546-0.40673664j  -0.80901699-0.58778525j
       -0.66913061-0.74314483j  -0.5       -0.8660254j
       -0.30901699-0.95105652j  -0.10452846-0.9945219j
        0.10452846-0.9945219j    0.30901699-0.95105652j
        0.5       -0.8660254j    0.66913061-0.74314483j
        0.80901699-0.58778525j   0.91354546-0.40673664j
        0.9781476 -0.20791169j   1.        +0.j        ]
```

inverse_real_fft(data, n=None, axis=-1)

Will return the inverse FFT of the real valued array `data`.

fft2d(data, s=None, axes=(-2,-1))

Will return the 2-dimensional FFT of the array `data`.

real_fft2d(data, s=None, axes=(-2,-1))

Will return the 2d FFT of the real valued array `data`.

C API

The interface to the FFTPACK library is performed via the fftpackmodule module, which is responsible for making sure that the arrays sent to the FFTPACK routines are in the right format (contiguous memory locations, right numerical storage format, etc). It provides interfaces to the following FFTPACK routines, which are also the names of the Python functions:

- `cffti(i)`
- `cfftf(data, savearea)`
- `cfftb(data, savearea)`
- `rffti(i)`
- `rfftf(data, savearea)`
- `rfftb(data, savearea)`

The routines which start with `c` expect arrays of complex numbers, the routines which start with `r` expect real numbers only. The routines which end with `i` are the initalization functions, those which end with `f` perform the forward FFTs and those which end with `b` perform the backwards FFTs.

The initialization functions require a single integer argument corresponding to the size of the dataset, and returns a work array. The forward and backwards FFTs require two array arguments -- the first is the data array, the second is the work array returned by the initialization function. They return arrays corresponding to the coefficients of the FFT, with the first element in the returned array corresponding to the DC component, the second one to the first fundamental, etc.The length of the returned array is 1 + half the length of the input array in the case of real FFTs, and the same size as the input array in the case of complex data.

```
>>> x = cos(arange(30.0)/30.0*2*pi)
>>> w = rffti(30)
>>> f = rfftf(x, w)
>>> f[0]
(-1+0j)
>>> f[1]
(13.6940664103+2.91076367145j)
>>> f[2]
```

```
(-0.913545457643-0.406736643076j)
```

Compilation Notes

On some platforms, precompiled optimized versions of the FFTPACK library are preinstalled on the operating system, and the compilation procedure needs to be modified to force the fftpackmodule file to be linked against those rather than the fftpacklite.c file which is shipped with NumPy.

15. LinearAlgebra Reference

The LinearAlgebra.py module provides a simple interface to the low-level linear algebra routines provided by either the LAPACK FORTRAN library or the compatible lapack_lite C library.

Python Interface

solve_linear_equations(a, b)

This function solves a system of linear equations with a square non-singular matrix a and a right-hand-side vector b. Several right-hand-side vectors can be treated simultaneously by making b a two-dimensional array (i.e. a sequence of vectors). The function inverse(a) calculates the inverse of the square non-singular matrix a by calling solve_linear_equations(a, b) with a suitable b.

inverse(a)

This function returns the inverse of the specified matrix a which must be square and non-singular. To within floating point precision, it should always be true that:

```
matrixmultiply(a, inverse(a)) == identity(len(a))
```

To test this claim, one can do e.g.:

```
>>> a = reshape(arange(25.0), (5,5)) + identity(5)
>>> print a
[[  1.   1.   2.   3.   4.]
 [  5.   7.   7.   8.   9.]
 [ 10.  11.  13.  13.  14.]
 [ 15.  16.  17.  19.  19.]
 [ 20.  21.  22.  23.  25.]]
>>> inv_a = inverse(a)
>>> print inv_a
[[ 0.20634921 -0.52380952 -0.25396825  0.01587302  0.28571429]
 [-0.5026455   0.63492063 -0.22751323 -0.08994709  0.04761905]
 [-0.21164021 -0.20634921  0.7989418  -0.1957672  -0.19047619]
 [ 0.07936508 -0.04761905 -0.17460317  0.6984127  -0.42857143]
 [ 0.37037037  0.11111111 -0.14814815 -0.40740741  0.33333333]]
>>> # Verify the inverse by printing the largest absolute element
... # of a * a^{-1} - identity(5)
... print "Inversion error:", \
... maximum.reduce(fabs(ravel(dot(a, inv_a)-identity(5))))
Inversion error: 2.6645352591e-015
```

eigenvalues(a)

This function returns the eigenvalues of the square matrix a.

```
>>> print a
[[ 1.  0.  0.  0.  0.]
 [ 0.  2.  0.  0.  1.]
```

```
 [ 0.  0.  3.  0.  0.]
 [ 0.  0.  0.  4.  0.]
 [ 0.  0.  0.  0.  1.]]
>>> print eigenvalues(a)
[ 1.  2.  3.  4.  1.]
```

eigenvectors(a)

This function returns both the eigenvalues and the eigenvectors, the latter as a two-dimensional array (i.e. a sequence of vectors).

```
>>> print a
[[ 1.  0.  0.  0.  0.]
 [ 0.  2.  0.  0.  1.]
 [ 0.  0.  3.  0.  0.]
 [ 0.  0.  0.  4.  0.]
 [ 0.  0.  0.  0.  1.]]
>>> evalues, evectors = eigenvectors(a)
>>> print evalues
[ 1.  2.  3.  4.  1.]
>>> print evectors
[[ 1.          0.          0.          0.          0.        ]
 [ 0.          1.          0.          0.          0.        ]
 [ 0.          0.          1.          0.          0.        ]
 [ 0.          0.          0.          1.          0.        ]
 [ 0.         -0.70710678  0.          0.          0.70710678]]
```

singular_value_decomposition(a, full_matrices=0)

This function returns three arrays V, S, and W^T whose matrix product is the original matrix a. V and W^T are unitary matrices (rank-2 arrays), whereas S is the vector (rank-1 array) of diagonal elements of the singular-value matrix. This function is mainly used to check whether (and in what way) a matrix is ill-conditioned.

generalized_inverse(a, rcond=1e-10)

This function returns the generalized inverse (also known as pseudo-inverse or Moore-Penrose-inverse) of the matrix a. It has numerous applications related to linear equations and least-squares problems.

determinant(a)

This function returns the determinant of the square matrix a.

linear_least_squares(a, b, rcond=e-10)

This function returns the least-squares solution of an overdetermined system of linear equations. An optional third argument indicates the cutoff for the range of singular values (defaults to 10-10). There are four return values: the least-squares solution itself, the sum of the squared residuals (i.e. the quantity minimized by the solution), the rank of the matrix a, and the singular values of a in descending order.

C API

Compilation Notes

On some platforms, precompiled optimized versions of the LAPACK library are preinstalled on the operating system, and the compilation procedure needs to be modified to force the lapackmodule.c file to be linked against those rather than the dlapack_lite.c and zlapack_lite.c files which are shipped with NumPy.

16. RandomArray Reference

The RandomArray.py module (in conjunction with the ranlibmodule.c file) provides a high-level interface to the ranlib module, which provides a good quality C implementation of a random-number generator.

Python Interface

seed(x=0, y=0)

The `seed()` function takes two integers and sets the two seeds of the random number generator to those values. If the default values of 0 are used for both x and y, then a seed is generated from the current time, providing a pseudo-random seed.

get_seed()

The `get_seed()` function returns the two seeds used by the current random-number generator. It is most often used to find out what seeds the seed() function chose at the last iteration. [thread-safety issue?]

random(shape=ReturnFloat)

The `random()` function takes a shape, and returns an array of double-precision floatings point numbers between 0.0 and 1.0. Neither 0.0 nor 1.0 is ever returned by this function. If no argument is specified, the function returns a single floating point number (not an array). The array is filled from the generator following the canonical array organization (see discussion of the `.flat` attribute)

uniform(minimum, maximum, shape=ReturnFloat)

The `uniform()` function returns an array of the specified shape and containing double-precision floating point random numbers strictly between minimum and maximum. If no shape is specified, a single number is returned.

randint(minimum, maximum, shape=ReturnFloat)

The `randint()` function returns an array of the specified shape and containing random (standard) integers greater than or equal to `minimum` and strictly less than `maximum`. If no shape is specified, a single number is returned.

permutation(n)

The `permutation()` function returns an array of the integers between `0` and `n-1`, in an array of shape `(n,)`, and with its elements randomly permuted.

An example use of the RandomArray module (exact output will be different each time!):

```
>>> from RandomArray import *
>>> seed()                      # Set seed based on current time
>>> print get_seed()            # Find out what seeds were used
(897800491, 192000)
>>> print random()
0.0528018975065
>>> print random((5,2))
[[ 0.14833829  0.99031458]
```

```
 [ 0.7526806   0.09601787]
 [ 0.1895229   0.97674777]
 [ 0.46134511  0.25420982]
 [ 0.66132009  0.24864472]]
>>> print uniform(-1,1,(10,))
[ 0.72168852 -0.75374185 -0.73590945  0.50488248 -0.74462822  0.09293685
      -0.65898308  0.9718067  -0.03252475  0.99611011]
>>> print randint(0,100, (12,))
[28  5 96 19  1 32 69 40 56 69 53 44]
>>> print permutation(10)
[4 2 8 9 1 7 3 6 5 0]
>>> seed(897800491, 192000)      # resetting the same seeds
>>> print random()               # yields the same numbers
0.0528018975065
```

C API

17. Glossary

This section will define a few of the technical words used throughout this document. [Please let us know of any additions to this list which you feel would be helpful -- the authors]

typecode: a single character describing the format of the data stored in an array. For example, 'b' refers to unsigned byte-sized integers (0-255).

ufunc / universal function: a ufunc is a callable object which performs operations on all of the elements of its arguments, which can be lists, tuples, or arrays. Many ufuncs are defined in the `umath` module.

array / multiarray: an array refers to the Python object type defined by the NumPy extensions to store and manipulate numbers efficiently.

UserArray: The UserArray module defines a UserArray class which should be subclassed by users wishing to have classes which behave similarly to the array object type.

Matrix: The Matrix module defines a subclass Matrix of the UserArray class which is specialized for linear algebra matrices. Most notably, it overrides the multiplication operator on Matrix instances to perform matrix multiplication instead of element-wise multiplication.

rank: the rank of an array is the number of dimensions it has, or the number of integers in its shape tuple.

shape: array objects have an attribute called shape which is necessarily a tuple. An array with an empty tuple shape is treated like a scalar (it holds one element).

18. Known Bugs and Limitations

This chapter lists the known bugs in the current version, which will be fixed in later releases, and lists known limitations, specifying whether they are likely to be addressed by later releases.

Bugs

Modify docstrings for zeros and ones to be right (shape tuple instead of multiple indices, and default typecode is int, not float.

Modify docstring for reshape to be right (shares data by default, exactly opposite of what's said).

Docstring for concatenate should be modified to include a tuple and optional axis

Should do typechecking early in arrayrange()

Weird:

```
>>> `arange(3) + ones((3,), 'i')`
'array([1, 2, 3])'
>>> `ones(3,), 'i') + arange(3)`
"array([1, 2, 3],'i')"
```

Limitations

Should define Precision.Char to be 'c', and come up with names for '1', 's' and 'i' (or is Int 'i')? Same F'?

array(2**8, Int0) and the like.

Int0 should be a 'b', not an Int8, no?

Int should be a native long int -- check?

Somewhere other than glossary, talk about rank-0 arrays

Get rid of extra notice that... note that...

add more examples w/ pictures, talk about arrays of strings and other gotcha's

Solicit ideas for two categories of other bits of knowledge to put somewhere:

- Gotcha's -- things to watch out for, such as the shared data bit, the uncontrolled upcasting, etc.
- Recipes -- combinations of functions which together do really neat things, such as Jim Hugunin's historgram function:
-

Index