

---

# Numarray

## An Open Source project

*Release 0.7*

Perry Greenfield  
Todd Miller  
Rick White  
J.C. Hsu  
Paul Barrett  
Jochen Küpper

Previously authored by:  
David Ascher  
Paul F. Dubois  
Konrad Hinsén  
Jim Hugunin  
Travis Oliphant

with contributions from the Numerical Python community

August 22, 2003

Space Telescope Science Institute, 3700 San Martin Dr, Baltimore, MD 21218  
UCRL-MA-128569

## Legal Notice

Please see file Legal.html in the source distribution.

This open source project has been contributed to by many people, including personnel of the Lawrence Livermore National Laboratory, Livermore, CA, USA. The following notice covers those contributions, including contributions to this manual.

Copyright (c) 1999, 2000, 2001. The Regents of the University of California. All rights reserved.

Permission to use, copy, modify, and distribute this software for any purpose without fee is hereby granted, provided that this entire notice is included in all copies of any software which is or includes a copy or modification of this software and in all copies of the supporting documentation for such software.

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.

## Special license for package MA

The package MA was written by Paul Dubois, Lawrence Livermore National Laboratory, Livermore, CA, USA.

Copyright (c) 1999, 2000. The Regents of the University of California. All rights reserved.

Permission to use, copy, modify, and distribute this software for any purpose without fee is hereby granted, provided that this entire notice is included in all copies of any software which is or includes a copy or modification of this software and in all copies of the supporting documentation for such software.

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.

## Disclaimer

This software was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor the University of California nor any of their employees, makes any warranty, express or implied, or assumes any liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately-owned rights. Reference herein to any specific commercial products, process, or service by trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or the University of California. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or the University of California, and shall not be used for advertising or product endorsement purposes.



# CONTENTS

<b>I</b>	<b>Numerical Python</b>	<b>1</b>
<b>1</b>	<b>Introduction</b>	<b>5</b>
1.1	Where to get information and code . . . . .	6
1.2	Acknowledgments . . . . .	6
<b>2</b>	<b>Installing numarray</b>	<b>9</b>
2.1	Testing the Python installation . . . . .	9
2.2	Testing the Numarray Python Extension Installation . . . . .	9
2.3	Installing numarray . . . . .	10
2.4	At the SourceForge... . . . .	12
<b>3</b>	<b>High-Level Overview</b>	<b>13</b>
3.1	Numarray Objects . . . . .	13
3.2	Universal Functions . . . . .	14
3.3	Convenience Functions . . . . .	14
3.4	Differences between numarray and Numeric. . . . .	15
<b>4</b>	<b>Array Basics</b>	<b>17</b>
4.1	Basics . . . . .	17
4.2	Creating arrays from scratch . . . . .	18
4.3	Creating arrays with values specified “on-the-fly” . . . . .	23
4.4	Coercion and Casting . . . . .	28
4.5	Operating on Arrays . . . . .	29
4.6	Getting and Setting array values . . . . .	31
4.7	Slicing Arrays . . . . .	33
4.8	Index Arrays . . . . .	35
4.9	Exception Handling . . . . .	36
<b>5</b>	<b>Ufuncs</b>	<b>39</b>
5.1	What are Ufuncs? . . . . .	39
5.2	Which are the Ufuncs? . . . . .	42
<b>6</b>	<b>Pseudo Indices</b>	<b>47</b>
<b>7</b>	<b>Array Functions</b>	<b>49</b>
<b>8</b>	<b>Array Methods</b>	<b>61</b>
<b>9</b>	<b>Array Attributes</b>	<b>65</b>

<b>10 C extension API</b>	<b>67</b>
10.1 Accessing the numarray C-API . . . . .	67
10.2 Fundamental data structures . . . . .	68
10.3 High-level API . . . . .	71
10.4 Element-wise API . . . . .	75
10.5 One-dimensional API . . . . .	79
10.6 Numeric emulation API . . . . .	80
10.7 New numarray functions . . . . .	83
 <b>II Extension modules</b>	 <b>87</b>
<b>11 Convolution</b>	<b>89</b>
11.1 Convolution functions . . . . .	89
11.2 Global constants . . . . .	89
<b>12 Fast-Fourier-Transform</b>	<b>91</b>
12.1 Installation . . . . .	91
12.2 FFT Python Interface . . . . .	91
12.3 fftpack Python Interface . . . . .	92
<b>13 Linear Algebra</b>	<b>95</b>
13.1 Installation . . . . .	95
13.2 Python Interface . . . . .	95
<b>14 Masked Arrays</b>	<b>99</b>
14.1 What is a masked array? . . . . .	99
14.2 Installing and using MA . . . . .	99
14.3 Class MaskedArray . . . . .	100
14.4 MaskedArray Attributes . . . . .	108
14.5 MaskedArray Functions . . . . .	108
14.6 Helper classes . . . . .	112
14.7 Examples of Using MA . . . . .	114
<b>15 Random Numbers</b>	<b>117</b>
15.1 General functions . . . . .	117
15.2 Special random number distributions . . . . .	117
15.3 Examples . . . . .	119
 <b>A Glossary</b>	 <b>125</b>
<b>Index</b>	<b>127</b>

**Part I**

**Numerical Python**



NumArray (“numarray”) adds a fast multidimensional array facility to Python. This part contains all you need to know about “numarray” arrays and the functions that operate upon them.





# Introduction

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

Numarray is a set of extensions to the Python programming language which allows 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, and 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. One significant difference is that numarray objects must contain elements of homogeneous type, while standard Python sequences can contain elements of mixed type. Two-dimensional arrays differ from matrices primarily in the way multiplication is performed; 2-D arrays are multiplied element-by-element.

This is a reimplementation of the earlier Numeric module (aka numpy). For the most part, the syntax of numarray is identical to that of Numeric, although there are significant differences. The differences are primarily in new features. For Python 2.2 and later, the syntax is completely backwards compatible. See the High-Level Overview (chapter 3) for incompatibilities for earlier versions of Python. The reasons for rewriting Numeric and a comparison between Numeric and numarray are also described in chapter 3. Portions of the present document are almost word-for-word identical to the Numeric manual. It has been updated to reflect the syntax and behavior of numarray, and there is a new section ( 3.4) on differences between Numeric and numarray.

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 and uses too much space. Anything which we can do in numarray 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, numarray is simply the application of this experience to the Python language. Thus many of the operations described in numarray 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 numarray include the infamous APL family of languages, Basis, MATLAB, FORTRAN, S and S+, and others. This heritage will be obvious to users of numarray who already have experience with these other languages. This manual, 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.

This document is the “official” documentation for numarray. It is both a tutorial and the most authoritative source of information about numarray with the exception of the source code. The tutorial material will walk you through a set of manipulations of simple, small arrays of numbers. This choice was made because:

- 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.
- Every reader will have at least an intuition as to the meaning of the data and organization of image files.

All users of numarray, whether interested in image processing or not, are encouraged to follow the tutorial with a working numarray installation at their side, testing the examples, and, more importantly, transferring the understanding

gained by working on arrays to their specific domain. The best way to learn is by doing — the aim of this tutorial is to guide you along this "doing."

Here is what the rest of this part contains:

**Installing numarray** Chapter 2 provides information on testing Python, numarray, and compiling and installing numarray if necessary.

**High-Level Overview** Chapter 3 gives a high-level overview of the components of the numarray system as a whole.

**Array Basics** Chapter 4 provides a detailed step-by-step introduction to the most important aspect of numarray, the multidimensional array objects.

**Ufuncs** Chapter 5 provides information on universal functions, the mathematical functions which operate on arrays and other sequences elementwise.

**Pseudo Indices** Chapter 6 covers syntax for some special indexing operators.

**Array Functions** Chapter 7 is a catalog of each of the utility functions which allow easy algorithmic processing of arrays.

**Array Methods** Chapter 8 discusses the methods of array objects.

**Array Attributes** Chapter 9 presents the attributes of array objects.

**Writing a C extension to numarray** Chapter not included yet.

**C API Reference** Chapter not included yet.

**Optional Packages** Not included yet.

**Glossary** Appendix A gives a glossary of terms.

## 1.1 Where to get information and code

Numarray and its documentation are available at SourceForge ([sourceforge.net](http://sourceforge.net); SourceForge addresses can also be abbreviated as [sf.net](http://sf.net)). The main web site is: <http://numpy.sourceforge.net>. Downloads, bug reports, a patch facility, and releases are at the main project page, reachable from the above site or directly at: <http://sourceforge.net/projects/numpy> (see Numarray under "Latest File Releases"). The Python web site is <http://www.python.org>. Although numarray will have a full suite of add-on modules available (e.g. `fft`, `random`) there are initially none. Check <http://stsdas.stsci.edu/numarray/index.html> for an up-to-date status on available modules compatible with numarray.

## 1.2 Acknowledgments

Numerical Python was 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 the Numerical Python version of this document, incorporating contributions from Konrad Hinsen and Travis Oliphant, both of whom are major contributors to Numerical Python. The reimplementations of Numeric as numarray were done primarily by Perry Greenfield, Todd Miller, and Rick White, with some assistance from J.C. Hsu and Paul Barrett. Although numarray is almost a completely new implementation, it owes a great deal to the ideas, interface and behavior expressed in the Numeric implementation. It is not an overstatement to say that the existence of Numeric made the implementation of numarray far, far easier than it would otherwise have been. Since the source for the original Numeric module was moved to SourceForge, the numarray user community has

become a significant part of the process. Numeric/numarray illustrates the power of the open source software concept. Please send comments and corrections to this manual to [perry@stsci.edu](mailto:perry@stsci.edu), or to Perry Greenfield, 3700 San Martin Dr, Baltimore, MD 21218, U.S.A.



# Installing numarray

This chapter explains how to install and test numarray, 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 include installing and testing Python, the numarray extensions, and some tools and sample files used in the examples of this tutorial.

## 2.1 Testing the Python installation

The first step is to install Python if you haven't already. Python is available from the Python project page at <http://sourceforge.net/projects/python>. Click on the link corresponding to your platform, and follow the instructions described there. Unlike earlier versions of numarray, version 0.5 and later require Python version 2.2.2 at a minimum. When installed, starting Python by typing `python` at the shell or double-clicking on the Python interpreter should give a prompt such as:

```
Python 2.2 (#1, Mar 15 2002, 14:38:30) [C] on linux2
Type "help", "copyright", "credits" or "license" for more information.
```

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

## 2.2 Testing the Numarray Python Extension Installation

The standard Python distribution does not come as of this writing with the numarray Python extensions installed, but your system administrator may have installed them already. To find out if your Python interpreter has numarray installed, type `'import numarray'` at the Python prompt. You'll see one of two behaviors (throughout this document user input and python interpreter output will be emphasized as shown in the block below):

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

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

```
>>> import numarray
>>> numarray.__version__
'0.6'
```

indicating that you do. If you do, you can skip the next section and go ahead to section 2.4. If you don't, you have to get and install the numarray extensions as described in section 2.3.

## 2.3 Installing numarray

The release facility at SourceForge is accessed through the project page, <http://sourceforge.net/projects/numpy>. Click on the "Numarray" release and you will be presented with a list of the available files. The files whose names end in ".tar.gz" are source code releases. The other files are binaries for a given platform (if any are available).

It is possible to get the latest sources directly from our CVS repository using the facilities described at SourceForge. Note that while every effort is made to ensure that the repository is always "good", direct use of the repository is subject to more errors than using a standard release.

### 2.3.1 Installing on Unix

The source distribution should be uncompressed and unpacked as follows (for example):

```
gunzip numarray-0.6.tar.gz
tar xf numarray-0.6.tar
```

Follow the instructions in the top-level directory for compilation and installation. Note that there are options you must consider before beginning. Installation is usually as simple as:

```
python setup.py install
```

or:

```
python setupall.py install
```

if you want to install all additional packages, which include `numarray.convolve`, `numarray.fft`, `numarray.linear_algebra`, and `numarray.random_array`.

See `numarray-X.XX/Doc/INSTALL.txt` for the latest details.

**Important Tip** Just like all Python modules and packages, the numarray module can be invoked using either the 'import numarray' form, or the 'from numarray import ...' form. Because most of the functions we'll talk about are in the numarray module, in this document, all of the code samples will assume that they have been preceded by a statement:

```
>>> from numarray import *
```

Note the lowercase name in numarray as opposed to Numeric.

## 2.3.2 Installing on Windows

To install numarray, you need to be in an account with Administrator privileges. As a general rule, always remove (or hide) any old version of numarray before installing the next version.

We have tested Numarray on several Win-32 platforms including:

- Windows-XP-Pro-x86 (MSVC-6.0)
- Windows-NT-x86 (MSVC-6.0)
- Windows-98-x86 (MSVC-6.0)

### Installation from source

1. Unpack the distribution: (NOTE: You may have to download an "unzipping" utility)

```
C:\> unzip numarray.zip
C:\> cd numarray
```

2. Build it using the distutils defaults:

```
C:\numarray> python setup.py install
```

This installs numarray in C:\pythonXX where XX is the version number of your python installation, e.g. 20, 21, etc.

### Installation from self-installing executable

1. Click on the executable's icon to run the installer.
2. Click "next" several times. I have not experimented with customizing the installation directory and don't recommend changing any of the installation defaults. If you do and have problems, let us know.
3. Assuming everything else goes smoothly, click "finish".

### Testing your Installation

Once you have installed numarray, test it with:

```
C:\numarray> python
Python 2.2.2 (#18, Dec 30 2002, 02:26:03) [MSC 32 bit (Intel)] on win32
Type "copyright", "credits" or "license" for more information.
>>> import numarray.testall as testall
>>> testall.test()
numarray: (0, 994)
recarray: (0, 30)
chararray: (0, 157)
memmap: (0, 59)
```

Each line in the above output indicates that 0 of X tests failed. X grows steadily with each release, so the numbers shown above are probably not current.



## Installation on Cygwin

For an installation of numarray for python running on Cygwin, see section 2.3.1.

## 2.4 At the SourceForge...

The SourceForge project page for numarray is at <http://sourceforge.net/projects/numpy>. On this project page you will find links to:

**The Numpy Discussion List** You can subscribe to a discussion list about numarray using the project page at SourceForge. The list is a good place to ask questions and get help. Send mail to [numpy-discussion@lists.sourceforge.net](mailto:numpy-discussion@lists.sourceforge.net). Note that there is no numarray-discussion group, we share the list created by the numeric community.

**The Web Site** Click on "home page" to get to the Numarray Home Page, which has links to documentation and other resources, including tools for connecting numarray to Fortran.

**Bugs and Patches** Bug tracking and patch-management facilities is provided on the SourceForge project page.

**CVS Repository** You can get the latest and greatest (albeit less tested and trustworthy) version of numarray directly from our CVS repository.

**FTP Site** The FTP Site contains this documentation in several formats, plus maybe some other goodies we have lying around.

# 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.

Numarray makes available a set of universal functions (technically ufunc objects), used in the same way they were used in Numeric. These are discussed in some detail in chapter 5.

## 3.1 Numarray Objects

The array objects are generally homogeneous collections of potentially large numbers of numbers. All numbers in a numarray 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, however. There is a "mask array" package ("MA") for Numeric; however, at the present time this has not yet been implemented for numarray.

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, unless the array is explicitly resized using the resize function.

The shape of an array is 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. In Python terms, the shape of an array is a tuple of integers, one integer for each dimension that represents the extent in that dimension. The rank of an array is the number of dimensions along which it is defined. It can change throughout the life of the array. Thus, the rank is the length of the shape (except for rank 0). **Note:** This is not the same meaning of rank as in linear algebra.)

The type of an array is a description of the kind of element it contains. It determines the itemsize of the array. In contrast to Numeric, an array type in numarray is an instance of a NumericType class, rather than a single character code. However, it has been implemented in such a way that one may use aliases, such as 'u1', 'i1', 'i2', 'i4', 'f4', 'f8', etc., as well as the original character codes, to set array types. 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).

To put this in more familiar mathematical language: 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). It is possible to create a rank-0 array, one which has no data — it has no dimension along which it can be indexed.

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

```

>>> vector1 = array([1,2,3,4,5])
>>> 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]
 [1 9]]
# note that this is not the matrix
# multiplication of linear algebra

```

If this example does not work for you because it complains of an unknown name "array", you forgot to begin your session with

```

>>> from numpy import *

```

See section 2.3.1.

## 3.2 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:

```

>>> print sin([pi/2., pi/4., pi/6.])
[ 1.  0.70710678  0.5      ]
>>> 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 + 4 + 5

```

Ufuncs are covered in detail in "Ufuncs" on page 27.

## 3.3 Convenience Functions

The numpy 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.

```

>>> data = arange(10)                # 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 numarray arrays are described in chapter 7. See page 52 for more information about `where` and page 58 for information on `resize`.

### 3.4 Differences between numarray and Numeric.

This new version was developed for a number of reasons. To summarize, we regularly deal with large datasets and the new version gives us the capabilities that we feel are necessary for working with such datasets. In particular:

1. Avoid promotion of array types in expressions involving Python scalars (e.g., `2.*<Float32 array>` should not result in a `Float64` array).
2. Ability to use memory mapped files (largely implemented).
3. Ability to access fields in arrays of records as numeric arrays without copying the data to a new array.
4. Ability to reference byteswapped data or non-aligned data (as might be found in record arrays) without producing new temporary arrays.
5. Reuse temporary arrays in expressions when possible.
6. Provide more convenient use of index arrays (`put` and `take`).

A new implementation was decided upon since many of the existing Numeric developers agree that the existing implementation is not suitable for massive changes and enhancements.

This version has nearly the full functionality of the basic Numeric (only masked arrays and the `convolve` function are missing, and the use of the `ufunc` methods `reduce`, `accumulate`, and `outer` for complex types). No other libraries have been adapted to Numarray yet. In particular, the FFT and lapack modules are not yet available. *Numarray is not fully compatible with Numeric.* (But it is very similar in most respects).

The incompatibilities are listed below. The C interface is completely different, so existing C extensions will not work with numarray.

1. Coercion rules are different. Expressions involving scalars may not produce the same type of arrays.
2. Types are represented by Type Objects rather than character codes (though the old character codes may still be used as arguments to the functions).

3. For versions of Python prior to 2.2, arrays have no public attributes. Accessor functions must be used instead (e.g., to get shape for array `x`, one must use `x.getshape()` instead of `x.shape`). When using Python 2.2 or later, however, the attributes of Numarray are in fact available.

A further comment on type is appropriate here. In numarray, types are represented by type objects and not character codes. As with Numeric there is a module variable `Float32`, but now it represents an instance of a `FloatingType` class. For example, if `x` is a `Float32` array, `x.type()` will return a `FloatingType` instance associated with 32-bit floats (instead of using `x.typecode()` as is done in Numeric). So the following will not work in numarray:

```
>>> if x.typecode() == 'f':
```

rather, use:

```
>>> if x.type() == Float32:
```

(All examples presume “`from numarray import *`” has been used instead of “`import numarray`”, see section 2.3.1.) The advantage of the new scheme is that other kinds of tests become simpler. The type classes are hierarchical so one can easily test to see if the array is an integer array. For example:

```
>>> if isinstance(x.type(), IntegralType):
```

or:

```
>>> if isinstance(x.type(), UnsignedIntegralType):
```

# Array Basics

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

## 4.1 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 differences in nomenclature. Here is a brief definition of the terms used in this tutorial, and more or less consistently in the error messages of `numarray`.

The Python objects under discussion are formally called “NumArray” (or even more correct “`numarray`”) objects (N-dimensional arrays), but informally we'll just call them “array objects” or just “arrays”. These are different from the array objects defined in the standard Python `array` module (which is an older module designed for processing one-dimensional data such as sound files).

These array objects hold their data in a fixed length homogeneous (but not necessarily contiguous) block of elements, i.e. their elements all have the same C type (such as a 64-bit floating-point number). This is quite different from most Python container objects, which are variable length heterogeneous collections. (**Note:** Although Numeric supports arrays of python objects, this hasn't been implemented yet for `numarray`.)

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 `numarray` arrays which take some getting used to. The first is that by default, operations on arrays are performed elementwise.<sup>1</sup> 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 elementwise 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 `matrixmultiply` function will be useful. The second behavior which will catch many users by surprise is that certain operations, such as slicing, return arrays which are simply different views of the same data; that is, they will in fact share their data. This will be discussed at length when we have more concrete examples

---

<sup>1</sup>This is common to IDL behavior but contrary to Matlab behavior.

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.

## 4.2 Creating arrays from scratch

### 4.2.1 `array()` and types

**`array`**(*buffer=None, type=None, shape=None, copy=1, savespace=0, typecode=None*)

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` function takes several 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 data type; if it is omitted, as in the example above, Python tries to find the best type which can represent all the elements. The optional second argument is the shape to use for the array.

Since the elements we gave our example were two floats and one integer, it chose `Float64` as the type of the resulting array. 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 all of its input 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], type=Float32)  # not the default type
>>> print a
[ 1.  2.  3.]
```

**`inputarray`**(*seq, type=None, typecode=None*)

This function converts scalars, lists and tuples to a `numarray`, if at all possible. It passes `numarrays` through, making copies only to convert types. In any other case a `TypeError` is raised.

**`asarray`**(*seq, type=None, typecode=None*)

An alias for `inputarray`. This is deprecated and only provided for compatibility with `Numeric`.

**Important Tip** Pop Quiz: What will be the type of an array defined as follows:

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

Hint: `-3j` is an imaginary number.

Answer: `Complex64`

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 type argument to the array creator function (and indeed to any function which accepts a so-called type for arrays) are:

1. Elements that can have values true or false: `Bool`.
2. Unsigned numeric types: `UInt8`, `UInt16`, `UInt32`, and `UInt64`<sup>1</sup>.
3. Signed numeric types:
  - Signed integer choices: `Int8`, `Int16`, `Int32`, `Int64`.
  - Floating point choices: `Float32`, `Float64`.
4. Complex number types: `Complex32`, `Complex64`.

To specify a type, e.g. `UInt8`, etc, the easiest method is just to specify it as a string:

```
a = array([10], type = 'UInt8')
```

Another approach is to directly access the type object from the numeric types module:

```
import numerictypes
a = array([10], type = numerictypes.UInt8)
```

The various means for specifying types are defined in table 4.1, with each item in a row being equivalent. The *preferred* methods are in the first 3 columns: numarray type object, type string, or type code. The last two columns were added for backwards compatability with Numeric and are not recommended for new code. Numarray type object and string names denote the size of the type in bits. The numarray type code names denote the size of the type in bytes. The type objects must be imported from or referenced via the `numerictypes` module. All type strings and type codes are specified using ordinary Python strings, and hence don't require an import. Complex type names denote the size of one component, real or imaginary, in bits/bytes.

Table 4.1: Type specifiers

Numarray Type	Numarray String	Numarray Code	Numeric String	Numeric Code
<code>Int8</code>	<code>'Int8'</code>	<code>'i1'</code>	<code>'Byte'</code>	<code>'l'</code>
<code>UInt8</code>	<code>'UInt8'</code>	<code>'u1'</code>	<code>'UByte'</code>	
<code>Int16</code>	<code>'Int16'</code>	<code>'i2'</code>	<code>'Short'</code>	<code>'s'</code>
<code>UInt16</code>	<code>'UInt16'</code>	<code>'u2'</code>	<code>'UShort'</code>	
<code>Int32</code>	<code>'Int32'</code>	<code>'i4'</code>	<code>'Int'</code>	<code>'i'</code>
<code>UInt32</code>	<code>'UInt32'</code>	<code>'u4'</code>	<code>'UInt'</code>	<code>'u'</code>
<code>Int64</code>	<code>'Int64'</code>	<code>'i8'</code>		
<code>UInt64</code> <sup>1</sup>	<code>'UInt64'</code>	<code>'u8'</code>		
<code>Float32</code>	<code>'Float32'</code>	<code>'f4'</code>	<code>'Float'</code>	<code>'f'</code>
<code>Float64</code>	<code>'Float64'</code>	<code>'f8'</code>	<code>'Double'</code>	<code>'d'</code>
<code>Complex32</code>	<code>'Complex32'</code>	<code>'c8'</code>		<code>'F'</code>
<code>Complex64</code>	<code>'Complex64'</code>	<code>'c16'</code>	<code>'Complex'</code>	<code>'D'</code>

## 4.2.2 Multidimensional Arrays

The following example shows one way of creating multidimensional arrays:

---

<sup>1</sup> `UInt64` is unsupported on Windows



```
>>> 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 type we wished:

```
>>> ma_floats = array([[1,2,3],[4,5,6]], type=Float32)
>>> 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 an attribute which is a tuple of integers giving the shape. The `getshape` method returns this tuple. In general, one can directly use the `shape` attribute (but only for Python 2.2 and later) to get or set its value. Since it isn’t supported for earlier versions of Python, subsequent examples will use `getshape` and `setshape` only. So, in this case:

```
>>> print ma.shape                # works only with Python 2.2 or later
>>> print ma.getshape()          # works with all Python versions
(2, 3)
```

Using the earlier definitions, this is a shape of rank 2, where the first axis has length 2, and the second axis has length 3. The rank of an array *A* is always equal to `len(A.getshape())`. Note that `shape` is an attribute and `getshape` is a method of array objects. They are the first of several that 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, and methods are functions that operate on 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.

### **reshape**(*a*, *shape*)

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])
>>> print a
[1 2 3 4 5 6 7 8]
>>> b = reshape(a, (2,4))           # 2*4 == 8
>>> print b
[[1 2 3 4]
 [5 6 7 8]]
>>> 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 `numarray` 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)`. There is also a `setshape` method, which changes the shape of an array in-place (see below).

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.getshape()
[[ 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, but it is an internal attribute. You can change the shape of an array by calling the `setshape` method (or by assigning a tuple to the shape attribute, in Python 2.2), which assigns a new shape to the array:

```

>>> a = array([1,2,3,4,5,6,7,8,9,10])
>>> a.getshape()
(10,)
>>> a.setshape((2,5))
>>> a.shape = (2,5) # for Python 2.2 and later
>>> print a
[[ 1  2  3  4  5]
 [ 6  7  8  9 10]]
>>> a.setshape((10,1)) # second axis has length 1
>>> print a
[[ 1]
 [ 2]
 [ 3]
 [ 4]
 [ 5]
 [ 6]
 [ 7]
 [ 8]
 [ 9]
[10]]
>>> a.setshape((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.setshape((6,-1))
Traceback (innermost last):
  File "<stdin>", line 1, in ?
ValueError: New shape is not consistent with the old shape

```

### For Advanced Users: Printing arrays

Sections denoted “For Advanced Users” will be used to indicate aspects of the functions which may not be needed for a first introduction at `numarray`, but which should be mentioned for the sake of completeness.

The default printing routine provided by the `numarray` 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.

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 `concatenate` function 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
[[0 1 2 3 4 5 6 7]
 [0 1 2 3 4 5 6 7]]
>>> print b.getshape()
(2, 8)

```

#### **resize**(*base, shape*)

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 correspond 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]]

```

**For Advanced Users** The `numarray` constructor takes a data argument, an optional type, and an optional shape argument. If the data argument is a sequence, then array creates a new object of type `numarray`, 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.

## 4.3 Creating arrays with values specified “on-the-fly”

#### **zeros**(*shape, type*)

#### **ones**(*shape, type*)

Often, one needs to manipulate arrays filled with numbers which aren’t available beforehand. The `numarray` 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), Float32)
>>> print o
[[ 1.  1.  1.]
 [ 1.  1.  1.]]

```

**`arrayrange`**(*first*, *limit=None*, *stride=1*, *type=None*, *shape=None*)

**`arange`**(*first*, *limit=None*, *stride=1*, *type=None*, *shape=None*)

The `arange` function is similar to the `range` function in Python, except that it returns an array as opposed to a list. `arange` and `arrayrange` are equivalent.

```

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

```

Combining the `arange` with the `reshape` function, we can get:

```

>>> big = reshape(arange(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]]

```

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

```

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

```

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

```

>>> print arange(5.0)
[ 0.  1.  2.  3.  4.]
>>> print arange(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 *type* argument can force the type of the resulting array, which is otherwise the “highest” of the starting and stopping arguments. The starting argument defaults to 0 if not specified. Note that if a type is specified which is “lower” than that which `arange` 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.)

#### 4.3.1 Creating an array from a function

**fromfunction**(*object*, *shape*)

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 (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]]
>>> 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 with arguments given by 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. Thus, the following will not work as expected:

```

>>> def buggy(test):
...     if test > 4: return 1
...     else: return 0
...
>>> print fromfunction(buggy, (10,))
1

```

Here is how to do it properly. We add a print statement to the function for clarity:

```

>>> def notbuggy(test):                # only works in Python 2.1 & later
...     print test
...     return where(test>4,1,0)
...
>>> fromfunction(notbuggy,(10,))
[0 1 2 3 4 5 6 7 8 9]
array([0, 0, 0, 0, 0, 0, 1, 1, 1, 1])

```

We leave it as an exercise for the reader to figure out why the “buggy” example gave the result 1.

#### **identity**(*size*)

The `identity` function takes a single integer argument and returns a square identity array (in the “matrix” sense) 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]]

```



## 4.4 Coercion and Casting

We’ve mentioned the types of arrays, and how to create arrays with the right type, but we haven’t covered what happens when arrays with different types interact. For some operations, the behavior of `numarray` is significantly different from `Numeric`.

### 4.4.1 Automatic Coercions and Binary Operations

In `numarray` (in contrast to `Numeric`), there is now a distinction between how coercion is treated in two basic cases: array/scalar operations and array/array operations. In the array/array case, the coercion rules are nearly identical to those of `Numeric`, the only difference being combining signed and unsigned integers of the same size. The array/array result types are enumerated in table 4.2.

Table 4.2: Array/Array Result Types

	Bool	Int8	UInt8	Int16	UInt16	Int32	UInt32	Int64	UInt64	Float32	Float64	Complex32	Complex64
Bool	Int8	Int8	UInt8	Int16	UInt16	Int32	UInt32	Int64	UInt64	Float32	Float64	Complex32	Complex64
Int8		Int8	Int16	Int16	Int32	Int32	Int64	Int64	Int64	Float32	Float64	Complex32	Complex64
UInt8			UInt8	Int16	UInt16	Int32	UInt32	Int64	UInt64	Float32	Float64	Complex32	Complex64
Int16				Int16	Int32	Int32	Int64	Int64	Int64	Float32	Float64	Complex32	Complex64
UInt16					UInt16	Int32	UInt32	Int64	UInt64	Float32	Float64	Complex32	Complex64
Int32						Int32	Int64	Int64	Int64	Float32	Float64	Complex32	Complex64
UInt32							UInt32	Int64	UInt64	Float32	Float64	Complex32	Complex64
Int64								Int64	UInt64	Float32	Float64	Complex32	Complex64
UInt64									UInt64	Float32	Float64	Complex32	Complex64
Float32										Float32	Float64	Complex32	Complex64
Float64											Float64	Complex32	Complex64
Complex32												Complex32	Complex64
Complex64													Complex64

Scalars, however, are treated differently. If the scalar is of the same “kind” as the array (for example, the array and scalar are both integer types) then the output is the type of the array, even if it is of a normally “lower” type than the scalar. Adding an `Int16` array with an integer scalar results in an `Int16` array, not an `Int32` array as is the case in `Numeric`. Likewise adding a `Float32` array to a float scalar results in a `Float32` array rather than a `Float64` array as is the case with `Numeric`. Adding an `Int16` array and a float scalar will result in a `Float64` array, however, since the scalar is of a higher kind than the array. 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 type of the array.

```
>>> print (array ((1, 2, 3), type=Int16) * 2).type()
numarray type: Int16
>>> 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 results of array/scalar operations are enumerated in table 4.3. Entries marked with “” are identical to their neighbors on the same row.

Table 4.3: Array/Scalar Result Types

	Bool	Int8	UInt8	Int16	UInt16	Int32	UInt32	Int64	UInt64	Float32	Float64	Complex32	Complex64
int	Int32	Int8	UInt8	Int16	UInt16	Int32	UInt32	Int64	UInt64	Float32	Float64	Complex32	Complex64
long	Int32	Int8	UInt8	Int16	UInt16	Int32	UInt32	Int64	UInt64	Float32	Float64	Complex32	Complex64
float	Float64	”	”	”	”	”	”	”	Float64	Float32	Float64	Complex32	Complex64
complex	Complex64	”	”	”	”	”	”	”	”	”	”	”	Complex64

## 4.4.2 The type value table

The type identifiers (`Float32`, etc.) are `NumericType` instances. The mapping between type and the equivalent C variable is machine dependent. The correspondences between types and C variables for 32-bit architectures are shown in Table 4.4.

Table 4.4: Type identifier table on a x86 computer.

# of bytes	# of bits	Identifier
1	8	Bool
1	8	Int8
1	8	UInt8
2	16	Int16
2	16	UInt16
4	32	Int32
4	32	UInt32
8	64	Int64
8	64	UInt64
4	32	Float32
8	64	Float64
8	64	Complex32
16	128	Complex64

## 4.4.3 Long: the platform relative type

The type identifier `Long` is aliased to either `Int32` or `Int64`, depending on the machine architecture where `numarray` is installed. On 32-bit platforms, `Long` is defined as `Int32`. On 64-bit (LP64) platforms, `Long` is defined as `Int64`. `Long` is used as the default integer type for arrays and for index values, such as those returned by `nonzero`.

## 4.4.4 Deliberate casts (potentially down)

### `astype(type)`

You may also force `numarray` 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 `UInt8`) and convert it to a 64-bit float, one can do:

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

The *type* can be any of the number types, “larger” or “smaller”. If it is larger, this is a cast-up. 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(Int32)
array([0, 0, 0, 1, 1])
```

If the *type* used with `astype` is the original array’s type, then a copy of the original array is returned.

## 4.5 Operating on Arrays

---

<sup>10</sup>`Float64`

<sup>20</sup>`Complex64`

### 4.5.1 Simple operations

If you have a keen eye, you have noticed that some of the previous examples did something new: they 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)
[ 0.84147096  0.90929741  0.141112 ]
>>> 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 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 ?
ValueError: Arrays have incompatible shapes
```

This is because there is no reasonable way for `numarray` 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.getshape()
(3,)
>>> b.getshape()
(4,3)

```

Note that the last axis of a is the same length as that of b (i.e. compare the last elements in their shape tuples). Because a's and b's last dimensions both have length 3, those two dimensions were “matched”, and a new dimension was created and automatically “assumed” for array a. The data already in a were “replicated” as many times as needed (4, in this case) to make the shapes of the two operand arrays conform. This replication (broadcasting) occurs when arrays are operands to binary operations and their shapes differ, based on the following algorithm:

- starting from the last axis, the axis lengths (dimensions) of the operands are compared,
- if both arrays have axis lengths greater than 1, but the lengths differ, 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.

## 4.5.2 In-place operations

Beginning with Python 2.0, Python supports the in-place operators `+=`, `-=`, `*=`, and `/=`. Numarray supports these operations, but you need to be careful. The right-hand side should be of the same type. Some violation of this is possible, but in general contortions may be necessary for using the smaller “kinds” of types.

```

>>> x = array ([1, 2, 3], type=Int16)
>>> x += 3.5
>>> print x
[4 5 6]

```

This area clearly needs improvement.

## 4.6 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 = arange(10)
>>> print a[0]                # get first element
0
>>> print a[1:5]              # get second through fifth elements
[1 2 3 4]
>>> print a[-1]               # get last element
9
>>> print a[:-1]              # get all but last element
[0 1 2 3 4 5 6 7 8]
```

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

```
>>> a = arange(9)
>>> a.setshape((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 element at first row, first column
0
>>> print a[0,1]              # get element at first row, second column
1
>>> print a[1,0]              # get element at second row, first column
3
>>> print a[2,-1]             # get element 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]]
```

## 4.7 Slicing Arrays

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

```
>>> a = reshape(arange(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”. (Actually, there is an important distinction; see below.) 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. There is one important distinction between slicing arrays and slicing standard Python sequence objects. A slice of a `list` is a new copy of that subset of the `list`; a slice of an array is just a view into the data of the first array. To force a copy, you can use the `copy` method. For example:

```
>>> a = arange(20)
>>> b = a[3:8]
>>> c = a[3:8].copy()
>>> a[5] = -99
>>> print b
[ 3  4 -99  6  7]
>>> print c
[3 4 5 6 7]
```

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(arange(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, :].

## 4.8 Index Arrays

Arrays used as subscripts have special meanings which implicitly invoke the functions `put` (page 50), `take` (page 49), or `compress` (page 53). If the array is of `Bool` type, then the indexing will be treated as the equivalent of the `compress` function. If the array is of an integer type, then a `take` or `put` operation is implied. We will generalize the existing `take` and `put` as follows: If *ind1*, *ind2*, ... *indN* are index arrays (arrays of integers whose values indicate the index into another array), then `x[ind1, ind2]` forms a new array with the same shape as *ind1*, *ind2* (they all must be broadcastable to the same shape) and values such: `result[i,j,k] = x[ind1[i,j,k], ind2[i,j,k]]`. In this example, *ind1*, *ind2* are index arrays with 3 dimensions (but they could have an arbitrary number of dimensions). To illustrate with some specific examples:

```
>>> # simple index array example
>>> x = 2*arange(10)
>>> ind = array([3,6,2,4,4])
>>> x[ind]
array([ 6, 12,  4,  8,  8])
>>> # index a 2-d array
>>> x = arange(12)
>>> x.setshape((3,4))
>>> x
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
>>> ind1 = array([2,1])
>>> ind2 = array([0,2])
>>> x[ind1, ind2]
array([8, 6])
>>> # multidimensional index arrays
>>> ind1 = array([[2,2],[1,0]])
>>> ind2 = array([[2,1],[0,1]])
>>> x[ind1, ind2]
array([[10,  9],
       [ 4,  1]])
>>> # Mindblowing combination of multidimensional index arrays with
>>> # partial indexing. Strap on your seatbelts.
>>> x[ind1]
array([[[ 8,  9, 10, 11],
        [ 8,  9, 10, 11]],
       [[ 4,  5,  6,  7],
        [ 0,  1,  2,  3]])
```

Note that in this last example, each index in the single index array (*ind1*) is treated as though *x* were given only one index. For each of these “single” indices, a 1-d array is returned, thus the combination of the 2 dimensions in the index array combined with the leftover dimension in the array being indexed produces a 3 dimensional array.

When using constants for some of the index positions, then the result uses that constant for all values. Slices and strides (at least initially) will not be permitted in the same subscript as index arrays. So

```
>>> x[ind1, 2]
array([[10, 10],
       [ 6,  2]])
```

would be legal, but



```
>>> x[ind1, 1:3]
Traceback (most recent call last):
[...]
    raise IndexError("Cannot mix arrays and slices as indices")
IndexError: Cannot mix arrays and slices as indices
```

would not be. Similarly for assignment:

```
x[ind1, ind2, ind3] = values
```

will form a new array such that:

```
x[ind1[i,j,k], ind2[i,j,k], ind3[i,j,k]] = values[i,j,k]
```

The index arrays and the value array must be broadcast consistent. (As an example: `ind1.setshape((5,4))`, `ind2.setshape((5,))`, `ind3.setshape((1,4))`, and `values.setshape((1,))`.)

```
# Index put example, using broadcasting and illustrating that Python
# integer sequences work as indices also.
>>> x = zeros((10,10))
>>> x[[2,5,6],array([0,1,9,3])[:,NewAxis]] = 111
>>> x
array([[ 0,  0,  0,  0,  0,  0,  0,  0,  0,  0],
       [ 0,  0,  0,  0,  0,  0,  0,  0,  0,  0],
       [111, 111,  0, 111,  0,  0,  0,  0,  0, 111],
       [ 0,  0,  0,  0,  0,  0,  0,  0,  0,  0],
       [ 0,  0,  0,  0,  0,  0,  0,  0,  0,  0],
       [111, 111,  0, 111,  0,  0,  0,  0,  0, 111],
       [111, 111,  0, 111,  0,  0,  0,  0,  0, 111],
       [ 0,  0,  0,  0,  0,  0,  0,  0,  0,  0],
       [ 0,  0,  0,  0,  0,  0,  0,  0,  0,  0],
       [ 0,  0,  0,  0,  0,  0,  0,  0,  0,  0]])
```

If indices are repeated, the last value encountered will be stored. When index values are out of range they will be clipped to the appropriate range. That is to say, an index that is too large will be taken to refer to the last element, and a negative index will be interpreted as zero (future versions will interpret negative indices in the usual Python style, counting backwards from the end). Use of the equivalent `take` and `put` functions will allow other interpretations of the indices (raise exceptions for out of bounds indices, allow negative indices to work backwards as they do when used individually, or for indices to wrap around). The same behavior applies for functions such as `choose` and `where`.

```
>>> x = 2*arange(10)
>>> x[[0, 5, 100, 5, -2]] = [1000, 1005, 1100, 2005, 3005]
>>> x
array([3005,  2,  4,  6,  8, 2005, 12, 14, 16, 1100])
```

## 4.9 Exception Handling

We desired better control over exception handling than currently exists in Numeric. This has traditionally been a problem area (see the numerous posts in [comp.lang.python](#) regarding floating point exceptions, especially those by

Tim Peters). Numeric raises an exception for integer computations that result in a divide by zero or multiplications that result in overflows. The exception is raised after that operation has completed on all the array elements. No exceptions are raised for floating point errors (divide by zero, overflow, underflow, and invalid results), the compiler and processor are left to their default behavior (which is usually to return Infs and NaNs as values).

The approach for numarray is to provide customizable error handling behavior. It should be possible to specify three different behaviors for each of the four error types independently. These are:

- Ignore the error.
- Print a warning.
- Raise a Python exception.

The current implementation does that and has been tested successfully on Windows, Solaris, Redhat and Tru64. The implementation uses the floating point processor “sticky status flags” to detect errors. One can set the error mode by calling the error object’s `setMode` method. For example:

```
>>> numarray.Error.setMode(all="warn") # the default mode
>>> numarray.Error.setMode(dividebyzero="raise", underflow="ignore", invalid="warn")
```

The Error object can also be used in a stacking manner, by using the `pushMode` and `popMode` methods rather than `setMode`. For example:

```
>>> numarray.Error.getMode()
_NumErrorMode(overflow='warn', underflow='warn', dividebyzero='warn', invalid='warn')
>>> numarray.Error.pushMode(all="raise") # get really picky...
>>> numarray.Error.getMode()
_NumErrorMode(overflow='raise', underflow='raise', dividebyzero='raise', invalid='raise')
>>> numarray.Error.popMode() # pop and return the 'new' mode
_NumErrorMode(overflow='raise', underflow='raise', dividebyzero='raise', invalid='raise')
>>> numarray.Error.getMode() # verify the original mode is back
_NumErrorMode(overflow='warn', underflow='warn', dividebyzero='warn', invalid='warn')
```

Integer exception modes work the same way. Although integer computations do not affect the floating point status flag directly, our code checks the denominator for 0 for divides (in much the same way Numeric does) and then performs a floating point divide by zero to set the status flag (overflows are handled similarly). So even integer exceptions use the floating point status flags indirectly.



# Ufuncs

## 5.1 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 as 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 `numarray` module:

Table 5.1: Universal Functions, or ufuncs. 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.

<code>add (+)</code>	<code>subtract (-)</code>	<code>multiply (*)</code>	<code>divide (/)</code>
<code>remainder (%)</code>	<code>power (**)</code>	<i><code>arccos</code></i>	<i><code>arccosh</code></i>
<i><code>arcsin</code></i>	<i><code>arcsinh</code></i>	<i><code>arctan</code></i>	<i><code>arctanh</code></i>
<i><code>cos</code></i>	<i><code>cosh</code></i>	<i><code>tan</code></i>	<i><code>tanh</code></i>
<i><code>log10</code></i>	<i><code>sin</code></i>	<i><code>sinh</code></i>	<i><code>sqrt</code></i>
<i><code>absolute</code></i>	<i><code>fabs</code></i>	<i><code>floor</code></i>	<i><code>ceil</code></i>
<i><code>fmod</code></i>	<i><code>exp</code></i>	<i><code>log</code></i>	
<code>maximum</code>	<code>minimum</code>		
<code>greater (&gt;)</code>	<code>greater_equal (&gt;=)</code>	<code>equal (==)</code>	
<code>less (&lt;)</code>	<code>less_equal (&lt;=)</code>	<code>not_equal (!=)</code>	
<code>logical_or (or)</code>	<code>logical_xor</code>	<code>logical_not (not)</code>	<code>logical_and (and)</code>
<code>bitwise_or ( )</code>	<code>bitwise_xor (^)</code>	<code>bitwise_not (~)</code>	<code>bitwise_and (&amp;)</code>
<code>rshift (&gt;&gt;)</code>	<code>lshift (&lt;&lt;)</code>		

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.          0.84147096  0.90929741  0.14112      -0.7568025
 -0.95892429 -0.27941549  0.65698659  0.98935825  0.41211849]
```

A unary ufunc returns an array with the same shape as its argument array, but with each element replaced by the application of the function to that 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 methods 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])
```

### 5.1.1 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 can also be written as the following using the Ufunc form:

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

In both cases, a temporary array is created to store the results of the computation before it is finally copied into *dataset*. 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 at least one case:

- If the type of the target array is not that which would normally be computed, the operation will not coerce the array to the expected data type. (The result is done in the expected data type, but coerced back to the original array type.)
- Example:

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

---

<sup>1</sup>for Python-2.2.2 and up: `dataset *= 1.20` also works

The output array does not need to be the same variable as the input array. In `numarray`, in contrast to `Numeric`, the output array may have any type (automatic conversion is performed on the output).

## 5.1.2 Ufuncs have special methods

### **reduce**(*a*, *axis=0*)

If you don't know about the `reduce` command in Python, review section 5.1.3 of the Python Tutorial (<http://www.python.org/doc/current/tut/>). 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 elements 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, dim=1)
[10 30]
```

### **accumulate**(*a*)

The `accumulate` ufunc method is similar 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) # same as add.accumulate(a)[-1] w/o side effects on a
45
```

### **outer**(*a*, *b*)

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]]

```

#### **reduceat()**

The `reduceat` method of Numeric has not been implemented in `numarray`.

### 5.1.3 Ufuncs always return new arrays

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

## 5.2 Which are the Ufuncs?

Table 5.1 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:

### 5.2.1 Unary Mathematical Ufuncs

Unary 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

```

Complex arrays have a `conjugate` method, but in `numarray` (in contrast to `Numeric`) there is no conjugate ufunc. (There will be, though.)

### 5.2.2 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.

### 5.2.3 Logical Ufuncs

The “logical” ufuncs also perform their operations on arrays in elementwise fashion, just like the “mathematical” ones. Two are special (maximum and minimum) 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 and of type Bool: `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 2 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], type=Int8)
>>> y
array([4, 5, 6])
>>> x & y          # bitwise_and(x,y)
array([4, 5, 0])
>>> x | y          # bitwise_or(x,y)
array([7, 7, 6])
>>> x ^ y          # bitwise_xor(x,y)
array([3, 2, 6])
>>> ~ x           # bitwise_not(x)
array([-8, -8, -1], type=Int8)
```

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 greater(b, .3))
[[0 1 1 0 0]
 [0 0 1 1 1]
 [0 0 0 1 1]
 [1 0 0 0 0]
 [1 1 0 0 0]]

```

## 5.2.4 Comparisons

The comparison functions `equal`, `not_equal`, `greater`, `greater_equal`, `less`, and `less_equal` are invoked by the operators `==`, `!=`, `>`, `>=`, `<`, and `<=` respectively, but they can also be called directly as functions. Continuing with the preceding example,

```

>>> 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.

The operators and the comparison functions are not exactly equivalent. To compare an array `a` with an object `b`, if `b` can be converted to an array, the result of the comparison is returned. Otherwise, zero is returned. This means that comparing a list and comparing an array can return quite different answers. Since the functional forms such as `equal` will try to make arrays from their arguments, using `equal` can result in a different result than using `==`.

```

>>> a = array([1, 2, 3])
>>> b = [1, 2, 3]
>>> print a == 2
[0 1 0]
>>> print b == 2
0
>>> print equal(a, 2)
[0 1 0]
>>> print equal(b, 2)
[0 1 0]

```

### 5.2.5 Ufunc shorthands

Numarray 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
```



## Pseudo Indices

This 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. The scalar was converted to a rank-0 array which was then 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: Arrays have incompatible shapes
```

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 across the second axis of the second vector. One way to do this is to use the reshape function:

```

>>> a.getshape()
(2,)
>>> b.getshape()
(3,)
>>> b2 = reshape(b, (3,1))
>>> print b2
[[1]
 [2]
 [3]]
>>> b2.getshape()
(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.getshape()
(3,)
>>> c = b[:, NewAxis]
>>> print c
[[1]
 [2]
 [3]]
>>> c.getshape()
(3,1)

```

Why use such a pseudo-index over the `reshape` function or `setshape` 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,:]

```

## 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 numarray 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. See the description of index arrays in the Array Basics section for an alternative to `take` and `put`. `take` 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 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))        # out-of-range index
[[ 0  1  2  3  4]
 [ 0  1  2  3  4]]
```

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,), axis=1)          # first column
[[ 0]
 [ 5]
 [10]
 [15]]
>>> print take(a, (0,1), axis=1)        # first and second column
[[ 0  1]
 [ 5  6]
 [10 11]
 [15 16]]
>>> print take(a, (0,4), axis=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. This, as with many other facets of `numarray`, is best understood by experiment.

```

>>> 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, suppose `m51` is a 2-D array of type `UInt8` containing a greyscale image. We can create a table mapping the integers 0–255 to integers 0–255 using a “compressive nonlinearity”:

```

>>> table = (255 - arange(256)**2 / 256).astype(UInt8)

```

Then we can perform the `take()` operation

```

>>> m51b = take(table, m51)

```

#### **put(*a*, *indices*, *values*)**

`put` is the opposite of `take`. The values of the array *a* at the locations specified in *indices* are set to the corresponding value of *values*. The array *a* must be a contiguous array. The argument *indices* can be any integer sequence object with values suitable for indexing into the flat form of *a*. The argument *values* must be any sequence of values that can be converted to the type of *a*.

```

>>> x = arange(6)
>>> put(x, [2,4], [20,40])
>>> print x
[ 0  1 20  3 40  5]

```

Note that the target array *a* is not required to be one-dimensional. Since *a* is contiguous and stored in row-major order, the array indices can be treated as indexing *a*’s elements in storage order. The routine `put` is thus

equivalent to the following (although the loop is in C for speed):

```
ind = array(indices, copy=0)
v = array(values, copy=0).astype(a.type())
for i in len(ind): a.flat[i] = v[i]
```

**putmask**(*a*, *mask*, *values*)

**putmask** sets those elements of *a* for which *mask* is true to the corresponding value in *values*. The array *a* must be contiguous. The argument *mask* must be an integer sequence of the same size (but not necessarily the same shape) as *a*. The argument *values* will be repeated as necessary; in particular it can be a scalar. The array values must be convertible to the type of *a*.

```
>>> x=arange(5)
>>> putmask(x, [1,0,1,0,1], [10,20,30,40,50])
>>> print x
[10  1 30  3 50]
>>> putmask(x, [1,0,1,0,1], [-1,-2])
>>> print x
[-1  1 -1  3 -1]
```

Note how in the last example, the third argument was treated as if it were `[-1, -2, -1, -2, -1]`.

**transpose**(*a*, *axes=None*)

**transpose** takes an array *a* and returns a new array which corresponds to *a* with the order of axes specified by the second argument. The default corresponds to reversing the order of all the axes.

```
>>> 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]]
```

**repeat**(*a*, *repeats*, *axis=0*)

**repeat** takes an array *a* 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:

```
>>> print a
[[0 1 2]
 [3 4 5]]
>>> print repeat(a, 2*ones(a.shape[0]))
[[0 1 2]
 [0 1 2]
 [3 4 5]
 [3 4 5]]
>>> print repeat(a, 2*ones(a.shape[1]), 1)
[[0 0 1 1 2 2]
 [3 3 4 4 5 5]]
```



**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 indicated by the value of the corresponding element in *a*. Assume *a* is an array 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(a,b), (c,d))
```

**ravel**(*a*)

Returns the argument array *a* as a 1d array. It is equivalent to `reshape(a, (-1,))`. There is a `ravel` method which reshapes the array in-place. Unlike `a.ravel()`, however, the `ravel` function works with non-contiguous arrays.

```
>>> a=arange(25)
>>> a.setshape(5,5)
>>> a.transpose()
>>> a.iscontiguous()
0
>>> a
array([[ 0,  5, 10, 15, 20],
       [ 1,  6, 11, 16, 21],
       [ 2,  7, 12, 17, 22],
       [ 3,  8, 13, 18, 23],
       [ 4,  9, 14, 19, 24]])
>>> a.ravel()
Traceback (most recent call last):
...
TypeError: Can't reshape non-contiguous arrays
>>> ravel(a)
array([ 0,  5, 10, 15, 20,  1,  6, 11, 16, 21,  2,  7, 12, 17, 22,  3,
        8, 13, 18, 23,  4,  9, 14, 19, 24])
```

**nonzero**(*a*)

`nonzero` returns a tuple of arrays containing the indices of the elements in *a* that are nonzero.

```
>>> a = array([-1, 0, 1, 2])
>>> nonzero(a)
(array([0, 2, 3]),)
```

**where**(*condition*, *x*, *y*)

The `where` function creates an array whose values are those of *x* at those indices where *condition* is true, and those of *y* otherwise. The shape of the result is the shape of *condition*. The type of the result is determined by the types of *x* and *y*. Either *x* or *y* (or both) can be a scalar, which is then used for all appropriate elements of *condition*.

```
>>> where( arange(10) >= 5, 1, 2)
array([2, 2, 2, 2, 2, 1, 1, 1, 1, 1])
```

Starting with `numarray-0.6`, `where` supports a one parameter form that is equivalent to `nonzero` but reads better:

```
>>> where(arange(10) % 2)
(array([1, 3, 5, 7, 9]),) # indices where expression is true
```

Like `nonzero`, the one parameter form of `where` can be used to do array indexing:

```
>>> a = arange(10,20)
>>> a[ where( a % 2 ) ]
array([11, 13, 15, 17, 19])
```

Note that for array indices which are boolean arrays, using `where` is not necessary but is still OK:

```
>>> a[(a % 2) != 0]
array([11, 13, 15, 17, 19])
>>> a[where((a%2) != 0)]
array([11, 13, 15, 17, 19])
```

**`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*, *offset*=0, *axis1*=0, *axis2*=1)

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*, *offset*=0)

Returns the sum of the elements in *a* along the diagonal specified by *offset*.

```
>>> 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. The result of the numarray version of this function is not identical to that of Numeric because of differences in the way equal elements are sorted.

```
>>> 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*, *type*, *shape=None*)

Will return the array formed by the binary data given in *string* of the specified *type*. 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 **byteswap** method described on page 62. If *shape* is not specified, the created array will be one dimensional.

**fromfile**(*file*, *type*, *shape=None*)

If *file* is a string then it is interpreted as the name of a file which is opened and read. Otherwise, *file* must be a Python file object which is read as a source of binary array data. **fromfile** reads directly into the newly created array buffer with no intermediate string, but otherwise is similar to **fromstring**, treating the contents of the specified file as a binary data string.

**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). **Note:** Somebody who does more linear algebra really needs to do this function right some day.

**matrixmultiply**(*m1*, *m2*)

This function multiplies matrices or matrices and vectors as matrices rather than elementwise. Compare:

```

>>> print a
[[0 1 2]
 [3 4 5]]
>>> print b
[1 2 3]
>>> print a*b
[[ 0  2  6]
 [ 3  8 15]]
>>> print matrixmultiply(a,b)
[ 8 26]

```

**clip**(*m*, *m\_min*, *m\_max*)

The clip function creates an array with the same shape and type 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, type=Float32)
>>> print clip(a, 1.5, 7.5)
[1.5 1.5 2.  3.  4.  5.  6.  7.  7.5]

```

**indices**(*shape*, *type=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 *i*th 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.getshape()
(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.setshape((5,2,1))
>>> print x
[[[0]
  [1]]

 [[2]
  [3]]

 [[4]
  [5]]

 [[6]
  [7]]

 [[8]
  [9]]]
>>> y = swapaxes(x, 0, 2)
>>> print y.getshape()
(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)

`innerproduct` produces the inner product of arrays *a* and *b*. It is equivalent to `matrixmultiply(a, transpose(b))`.

**outerproduct**(*a*,*b*)

`outerproduct` produces the outer product of vectors *a* and *b*, that is `result[i, j] = a[i] * b[j]`.

**array\_repr**()

See section ?? on Textual Representations of arrays.

**array\_str**()

See section ?? 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*=0, *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.

**repeat**(*a*, *counts*, *axis*=0)

The `repeat` function uses repeated copies of *a* to create a result. The *axis* argument refers to the axis of *a* which will be replicated. The *counts* argument tells how many copies of each element to make. The length of *counts* must be the `len(a.getshape()[axis])`. In one dimension this is straightforward:

```
>>> y
array([0, 1, 2, 3, 4, 5])
>>> repeat(y, (1,2,0,2,2,3))
array([0, 1, 1, 3, 3, 4, 4, 5, 5, 5])
```

In more than one dimension it sometimes gets harder to understand. Consider for example this array *x* whose shape is (2,3).

```

>>> x
array([[0, 1, 2],
       [3, 4, 5]])
>>> repeat(x, (2,6))
array([[0, 1, 2],
       [0, 1, 2],
       [3, 4, 5],
       [3, 4, 5],
       [3, 4, 5],
       [3, 4, 5],
       [3, 4, 5],
       [3, 4, 5]])
>>> repeat(x, (6,3,2), 1)
array([[0, 0, 0, 0, 0, 0, 1, 1, 1, 2, 2],
       [3, 3, 3, 3, 3, 3, 4, 4, 4, 5, 5]])

```

### **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*, axis=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*, axis=0)**

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

### **product(*a*, axis=0)**

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

### **cumproduct(*a*, axis=0)**

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

### **alltrue(*a*, axis=0)**

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

### **sometrue(*a*, axis=0)**

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



**allclose**(*x*, *y*, *rtol* = 1.e-5, *atol* = 1.e-8)

This function tests whether or not arrays *x* and *y* of an integer or real type are equal subject to the given relative and absolute tolerances. The formula used is:

$$|x - y| < atol + rtol * |y| \quad (7.1)$$

This means essentially that both elements are small compared to *atol* or their difference divided by *y*'s value is small compared to *rtol*.

**See Also:**

[Module `numpy.convolve`](#) (section 11):

The `convolve` function is implemented in the optional `numpy.convolve` package.

[Module `numpy.correlate`](#) (section 11):

The `correlate` function is implemented in the optional `numpy.correlate` package.

## 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.

### **itemsizes()**

The `itemsizes` method applied to an array returns the number of bytes used by any one of its elements.

```
>>> a = arange(10)
>>> a.itemsizes()
4
>>> a = array([1.0])
>>> a.itemsizes()
8
>>> a = array([1], type=Complex64)
>>> a.itemsizes()
16
```

### **iscontiguous()**

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

```
>>> b = a[3:8:2]
>>> print a.iscontiguous()
1
>>> print b.iscontiguous()
0
```

### **copy()**

The `copy` method returns a copy of an array. When making an assignment or taking a slice, a new array object is created and has its own attributes, except that the data attribute just points to the data of the first array. The `copy` method may be used when it is important to obtain an independent copy. Another application was mentioned above, to obtain a contiguous array from a non-contiguous array.

### **type()**

The `type` method returns the type of the array it is applied to. While we've been talking about them as `Float32`, `Int16`, etc., it is important to note that they are not character strings, they are instances of `NumericType` classes. So this is what you'll get:



When using Python 2.2 or later, there are four public attributes which correspond to those of Numeric type objects. These are `shape`, `flat`, `real`, and `imag` (or `imaginary`). The following methods are provided as an alternative to using these attributes, either as a matter of preference or for those using versions of Python prior to 2.2.

**`getshape()`**

**`setshape()`**

The `getshape` method returns the tuple that gives the shape of the array. `setshape` assigns its argument (a tuple) to the internal attribute which defines the array shape. When using Python 2.2 or later, the `shape` attribute can be accessed or assigned to, which is equivalent to using these methods.

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

**`getflat()`**

The `getflat` method is equivalent to using the `flat` attribute of Numeric. For compatibility with Numeric, there is no `setflat` method, although the attribute can in fact be set using `setshape`.

```
>>> print a
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]]
>>> print a.getflat()
[ 0  1  2  3  4  5  6  7  8  9 10 11]
```

**`getreal()`**

**`setreal()`**

The `getreal` and `setreal` methods can be used to access or assign to the real part of an array containing imaginary elements.

**`getimag()`**

**`getimaginary()`**

**`setimag()`**

**`setimaginary()`**

The `getimag` and `setimag` methods can be used to access or assign to the imaginary part of an array containing imaginary elements. `getimaginary` is equivalent to `getimag`, and `setimaginary` is equivalent to `setimag`.

**`sum()`**

The `sum` method returns the sum of all elements in an array.

```
>>> arange(10).sum()
45
```

**`mean()`**

The `mean` method returns the average of all elements in an array.

```
>>> arange(10).mean()  
4.5
```

#### **min()**

The `min` method returns the smallest element in an array.

```
>>> arange(10).min()  
0
```

#### **max()**

The `max` method returns the largest element in an array.

```
>>> arange(10).max()  
9
```

## Array Attributes

There are four array attributes; however, they are only available when using Python 2.2 or later. There are array methods that may be used instead. The attributes are shape, flat, real and imaginary.

### shape

Accessing the shape attribute is equivalent to calling the `getshape` method; it returns the shape tuple. Assigning a value to the shape attribute is equivalent to calling the `setshape` method.

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

### flat

Accessing the flat attribute of an array returns the flattened, or raveled version of that array, without having to do a function call. This is equivalent to calling the `getflat` method. The returned array has the same number of elements as the input array, but it 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[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

### imag

## imaginary

These attributes exist only for complex arrays. They return respectively arrays filled with the real and imaginary parts of their elements. The equivalent methods for getting and setting these values are `getreal`, `setreal`, `methodgetimag` and `methodsetimag`. `methodgetimaginary` and `methodsetimaginary` are synonyms for `methodgetimag` and `methodsetimag` respectively, and `.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.type(), x.real.type()
D d
>>> print x.itemsize(), x.imag.itemsize()
16 8
```

# C extension API

This chapter describes the different available C-APIs for `numarray` based extension modules.

While this chapter describes the `numarray`-specifics for writing extension modules, a basic understanding of Python extension modules is expected. See Python's [Extending and Embedding](#) tutorial and the [Python/C API](#).

The `numarray` C-API has several different facets, and the first three facets each make different tradeoffs between memory use, speed, and ease of use. An additional facet provides backwards compatibility with legacy Numeric code. The final facet consists of miscellaneous function calls used to implement and utilize `numarray`, that were not part of Numeric.

In addition to most of the basic functionality provided by Numeric, these APIs provide access to misaligned, byteswapped, and discontinuous `numarrays`. Byteswapped arrays arise in the context of portable binary data formats where the byteorder specified by the data format is not the same as the host processor byte order. Misaligned arrays arise in the context of tabular data: files of records where arrays are superimposed on the column formed by a single field in the record. Discontinuous arrays arise from operations which permute the shape and strides of an array, such as `reshape`.

**high-level** This is the cleanest and easiest to use API. It creates temporary arrays to handle difficult cases (discontinuous, byteswapped, misaligned) in C code. Code using this API is written in terms of a pointer to a contiguous 1D array of C data. See section 10.3.

**element-wise** This API handles misbehaved arrays without creating temporaries. Code using this API is written to access single elements of an array via macros or functions. **Note:** These macros are probably slow, and the functions even slower. See section 10.4.

**1-dimensional** Code using this API get/sets consecutive elements of the inner dimension of an array, enabling the API to factor out tests for alignment and byteswapping to one test per call. Fewer tests means better performance, but at a cost of some temporary data and more difficult usage. See section 10.5.

**Numeric compatibility functions** This API aims to provide a reasonable (if not complete) emulation of the Numeric C-API. It is written in terms of the `numarray` high level API so that misbehaved `numarrays` are copied prior to processing with legacy Numeric code. See section 10.6.

**New `numarray` functions** This last facet of the C-API consists of function calls which have been added to `numarray` which are orthogonal to each of the 3 native access APIs and not part of the original Numeric. See section 10.7

## 10.1 Accessing the `numarray` C-API

There's a couple things you need to do in order to access `numarray`'s C-API in your own C extension module:



### 10.1.1 include numarray.h

Near the top of your extension module add the line:

```
#include "numarray.h"
```

This gives your C-code access to the numarray typedefs, macros, and function prototypes.

### 10.1.2 import libnumarray

In your extension module's initialization function, add the line:

```
import_libnumarray();
```

`import_libnumarray()` is actually a macro which sets up a pointer to the numarray C-API jump table. If you forget to call `import_libnumarray()`, your extension module will crash as soon as you call a numarray API function, because your application will attempt to dereference a NULL pointer.

Note that there is also a Numeric compatible API which substitutes `arrayobject.h` for `numarray.h` and `import_array()` for `import_libnumarray()` respectively.

## 10.2 Fundamental data structures

### 10.2.1 Numarray Numerical Data Types

Numarray hides the C implementation of its basic array elements behind a set of C typedefs which specify the absolute size of the type in bits. This approach enables a programmer to specify data items of arrays and extension functions in an explicit yet portable manner. In contrast, basic C types are platform relative, and so less useful for describing real physical data. Here are the names of the concrete Numarray element types:

- Bool
- Int8, UInt8
- Int16, UInt16
- Int32, UInt32
- Int64, UInt64
- Float32, Float64
- Complex32, Complex64

### 10.2.2 NumarrayType

The type of a numarray is communicated in C via one of the following enumeration constants. Type codes which are backwards compatible with Numeric are defined in terms of these constants, but use these if you're not already using the Numeric codes. These constants communicate type requirements between one function and another, since in C, you cannot pass a typedef as a value. `tAny` is used to specify both "no type requirement" and "no known type" depending on context.

```

typedef enum
{
    tAny,

    tBool,
    tInt8,      tUInt8,
    tInt16,     tUInt16,
    tInt32,     tUInt32,
    tInt64,     tUInt64,
    tFloat32,   tFloat64,
    tComplex32, tComplex64,

    tDefault = tFloat64,

#ifdef LP64
    tLong = tInt64
#else
    tLong = tInt32
#endif

} NumarrayType;

```

### 10.2.3 PyArray\_Descr

`PyArray_Descr` is used to hold a few parameters related to the type of an array and exists mostly for backwards compatibility with Numeric. *type\_num* is a `NumarrayType` value. *elsize* indicates the number of bytes in one element of an array of that type. *type* is a Numeric compatible character code.

Numarray's `PyArray_Descr` is currently missing the type-casting, `ones`, and `zeroes` functions. Extensions which use these missing Numeric features will not yet compile. Arrays of type `Object` are not yet supported.

```

typedef struct {
    int  type_num; /* PyArray_TYPES */
    int  elsize;   /* bytes for 1 element */
    char type;     /* One of "cblsilfdFD " Object arrays not supported. */
} PyArray_Descr;

```

### 10.2.4 PyArrayObject

The fundamental data structure of numarray is the `PyArrayObject`, which is named and laid out to maximize compatibility with Numeric. The numarray version of `PyArrayObject` is compile-time compatible with most but not all Numeric code. The constant `MAXDIM`, the maximum number of dimensions in an array, is typically defined as 40. It should be noted that unlike earlier versions of numarray, the present `PyArrayObject` structure is a first class python object, with full support for the number protocols in C. Well-behaved arrays have mutable fields which will reflect modifications back into Python“for free”.

```

typedef int maybelong;          /* towards 64-bit without breaking extensions. */

typedef struct {
    /* Numeric compatible stuff */

    PyObject_HEAD
    char *data;                 /* points to the actual C data for the array */
    int nd;                     /* number of array shape elements */
    maybelong dimensions[MAXDIM]; /* values of shape elements */
    maybelong strides[MAXDIM];   /* values of stride elements */
    PyObject *base;             /* unused, but don't touch! */
    PyArray_Descr *descr;       /* pointer to descriptor for this array's type */
    int flags;                  /* bitmask defining various array properties */

    /* numarray extras */

    PyObject *_data;            /* object must meet buffer API */
    PyObject *_shadows;         /* ill-behaved original array. */
    int nstrides;               /* elements in strides array */
    long byteoffset;            /* offset into buffer where array data begins */
    long bytestride;            /* basic separation of elements in bytes */
    long itemsize;              /* length of 1 element in bytes */

    char byteorder;             /* NUM_BIG_ENDIAN, NUM_LITTLE_ENDIAN */

    char _aligned;              /* test override flag */
    char _contiguous;           /* test override flag */

    /* Don't expect the following vars to stay around. Never use them.
       They're an implementation detail of the get/set macros. */

    Complex64 temp;             /* temporary for get/set macros */
    char * wptr;                /* working pointer for get/set macros */
} PyArrayObject;

```

## 10.2.5 Flag Bits

The following are the definitions for the bit values in the *flags* field of each numarray. Low order bits are Numeric compatible, higher order bits were added by numarray.

```

/* Array flags */
#define CONTIGUOUS      1      /* compatible, depends */
#define OWN_DIMENSIONS  2      /* always false */
#define OWN_STRIDES     4      /* always false */
#define OWN_DATA        8      /* always false */
#define SAVESPACE       0x10   /* not used */

#define ALIGNED         0x100   /* roughly: data % itemsize == 0 */
#define NOTSWAPPED      0x200   /* byteorder == sys.byteorder */
#define WRITABLE        0x400   /* data buffer is writable */

#define IS_CARRAY (CONTIGUOUS | ALIGNED | NOTSWAPPED)

```

## 10.3 High-level API

The high-level native API accepts an object (which may or may not be an array) and transforms the object into an array which satisfies a set of “behaved-ness requirements”. The idea behind the high-level API is to transparently convert misbehaved numarrays, ordinary sequences, and python scalars into C-arrays. A “misbehaved array” is one which is byteswapped, misaligned, or discontinuous. This API is the simplest and fastest, provided that your arrays are small. If you find your program is exhausting all available memory, it may be time to look at one of the other APIs.

### 10.3.1 High-level functions

The high-level support functions for interchanging numarrays between Python and C are as follows:

`PyArrayObject*` **NA\_InputArray** (*PyObject \*numarray, NumarrayType t, int requires*)

The purpose of `NA_InputArray` is to transfer array data from Python to C.

`PyArrayObject*` **NA\_OutputArray** (*PyObject \*numarray, NumarrayType t, int requires*)

The purpose of `NA_OutputArray` is to transfer data from C to Python. Practically speaking, the output numarray must be a `PyArrayObject`, and cannot be an arbitrary Python sequence.

`PyArrayObject*` **NA\_IoArray** (*PyObject \*numarray, NumarrayType t, int requires*)

`NA_IoArray` has fully bidirectional data transfer, creating the illusion of call-by-reference.

For a well-behaved array, there is no difference between the three, as no temporary is created and the returned object is identical to the original object (with an additional reference). For a mis-behaved input array, a well-behaved temporary will be created and the data copied from the original to the temporary. Since it is an input, modifications to its contents are not guaranteed to be reflected back to Python, and in the case where a temporary was created, won’t be. For a mis-behaved output array, any data side-effects generated by the C code will be safely communicated back to Python, but the initial numarray contents are undefined. For an I/O array, any required temporary will be initialized to the same contents as the original numarray, and any side-effects caused by C-code will be copied back to the original numarray. The array factory routines of the Numeric compatibility API are written in terms of `NA_IoArray`.

The return value of each function (`NA_InputArray`, `NA_OutputArray`, or `NA_IoArray`) is either a reference to the original numarray object, or a reference to a temporary numarray. Following execution of the C-code in the extension function body this pointer should *always* be DECREFed. When a temporary is DECREFed, it is deallocated, possibly after copying itself onto the original array. The one exception to this rule is that you should not DECREF an array returned via the `NA_ReturnOutput` function.

The *numarray* parameter specifies the original numarray object to be interfaced. Nested lists and tuples of numbers can be converted by `NA_InputArray` and `NA_IoArray` into a temporary numarray. The temporary is lost on function exit. Strictly speaking, allowing `NA_IoArray` to accept a list or tuple is a wart, since it will lose any side

effects. In principle however, communication back to lists and tuples can be supported even though it is not currently.

The `NumarrayType` *t* is an enumeration value which defines the type the array data should be converted to. Arrays of the same type are passed through unaltered, while mis-matched arrays are cast into temporaries of the specified type. The value `tAny` may be specified to indicate that the extension function can handle any type correctly.

The *requires* integer indicates under what conditions a temporary should be made. The simple way to specify it is to use `NUM_C_ARRAY`. This will cause the API function to make a well-behaved temporary if the original is byteswapped, misaligned, or discontinuous.

There is one other pair of high level function which serves to return output arrays as the function value: `NA_OptionalOutputArray` and `NA_ReturnOutput`.

`PyObject*` **`NA_OptionalOutputArray`**(*PyObject \*numarray, NumarrayType t, int requires, PyObject \*master*)

`NA_OptionalOutputArray` is essentially `NA_OutputArray`, but with one twist: if the original array *numarray* has the value `NULL` or `Py_None`, a temporary copy of *numarray master* is returned. This facilitates writing functions where the output array may or may-not be specified by the Python user.

`PyObject*` **`NA_ReturnOutput`**(*PyObject \*numarray, PyObject \*shadow*)

`NA_ReturnOutput` accepts as parameters both the original *numarray* and the value returned from `NA_OptionalOutputArray`, *shadow*. If *numarray* is `Py_None` or `NULL`, then *shadow* is returned. Otherwise, an output array was specified by the user, and `Py_None` is returned. This facilitates writing functions in the *numarray* style where the specification of an output array renders the function “mute”, with all side-effects in the output array and `None` as the return value.

### 10.3.2 Behaved-ness Requirements

Calls to the high level API specify a set of requirements that incoming arrays must satisfy. The requirements set is specified by a bit mask which is or’ed together from bits representing individual array requirements. An ordinary C array satisfies all 3 requirements: it is contiguous, aligned, and not byteswapped. It is possible to request arrays satisfying any or none of the behavedness requirements. Arrays which do not satisfy the specified requirements are transparently “shadowed” by temporary arrays which do satisfy them. By specifying `NUM_UNCONVERTED`, a caller is certifying that his extension function can correctly and directly handle the special cases possible for a `NumArray`, excluding type differences.

```
typedef enum
{
    NUM_CONTIGUOUS=1,
    NUM_NOTSWAPPED=2,
    NUM_ALIGNED=4,
    NUM_WRITABLE=8,

    NUM_C_ARRAY = (NUM_CONTIGUOUS | NUM_ALIGNED | NUM_NOTSWAPPED),
    NUM_UNCONVERTED = 0
}
```

### 10.3.3 Example

A C wrapper function using the high-level API would typically look like the following.<sup>1</sup>

<sup>1</sup>This function is taken from the `convolve` example in the source distribution.

```

static PyObject *
Py_Convolve1d(PyObject *obj, PyObject *args)
{
    PyObject *okernel, *odata, *oconvolved=Py_None;
    PyArrayObject *kernel, *data, *convolved;

    if (!PyArg_ParseTuple(args, "OO|O", &okernel, &odata, &oconvolved))
        return PyErr_Format(_convolveError,
                            "Convolve1d: Invalid parameters.");

```

First, define local variables and parse parameters. `Py_Convolve1d` expects two or three array parameters in `args`: the convolution kernel, the data, and optionally the return array. We define two variables for each array parameter, one which represents an arbitrary sequence object, and one which represents a `PyArrayObject` which contains a conversion of the sequence. If the sequence object was already a well-behaved `numarray`, it is returned without making a copy.

```

/* Align, Byteswap, Contiguous, Typeconvert */
kernel = NA_InputArray(okernel, tFloat64, C_ARRAY);
data = NA_InputArray(odata, tFloat64, C_ARRAY);
convolved = NA_OptionalOutputArray(oconvolved, tFloat64, C_ARRAY, data);

if (!kernel || !data || !convolved)
    return PyErr_Format(_convolveError,
                        "Convolve1d: error converting array inputs.");

```

These calls to `NA_InputArray` and `OptionalOutputArray` require that the arrays be aligned, contiguous, and not byteswapped, and of type `Float64`, or a temporary will be created. If the user hasn't provided a output array we ask `NA_OptionalOutputArray` to create a copy of the input `data`. We also check that the array screening and conversion process succeeded by verifying that none of the array pointers is `NULL`.

```

if ((kernel->nd != 1) || (data->nd != 1))
    return PyErr_Format(_convolveError,
                        "Convolve1d: arrays must have 1 dimension.");

if (!NA_ShapeEqual(data, convolved))
    return PyErr_Format(_convolveError,
                        "Convolve1d: data and output arrays need identical shapes.");

```

Make sure we were passed one-dimensional arrays, and data and output have the same size.

```

Convolve1d(kernel->dimensions[0], NA_OFFSETDATA(kernel),
            data->dimensions[0], NA_OFFSETDATA(data),
            NA_OFFSETDATA(convolved));

```

Call the C function actually performing the work. `NA_OFFSETDATA` returns the pointer to the first element of the array, adjusting for any byteoffset.

```

Py_XDECREF(kernel);
Py_XDECREF(data);

```

Decrease the reference counters of the input arrays. These were increased by `NA_InputArray`. `Py_XDECREF` tolerates NULL. `DECREF`'ing the `PyArrayObject` is how temporaries are released and in the case of IO and Output arrays, copied back onto the original.

```

    /* Align, Byteswap, Contiguous, Typeconvert */
    return NA_ReturnOutput(oconvolved, convolved);
}

```

Now return the results, which are either stored in the user-supplied array *oconvolved* and `Py_None` is returned, or if the user didn't supply an output array the temporary *convolved* is returned.

If your C function creates the output array you can use the following sequence to pass this array back to Python:

```

double *result;
int m, n;
.
.
.
result = func(...);
if(NULL == result)
    return NULL;
return NA_NewArray((void *)result, tFloat64, 2, m, n);
}

```

The C function `func` returns a newly allocated (m, n) array in *result*. After we check that everything is ok, we create a new `numarray` using `NA_NewArray` and pass it back to Python. `NA_NewArray` creates a `numarray` with `C_ARRAY` properties. If you wish to create an array that is byte-swapped, or misaligned, you can use `NA_NewAll`.

The C-code of the core convolution function is uninteresting. The main point of the example is that when using the high-level API, `numarray` specific code is confined to the wrapper function. The interface for the core function can be written in terms of primitive `numarray`/C data items, not objects. This is possible because the high level API can be used to deliver C arrays.

```

static void Convolve1d(long ksize, Float64 *kernel,
    long dsize, Float64*data, Float64 *convolved)
{
    long xc; long halfk = ksize/2;

    for(xc=0; xc<halfk; xc++)
        convolved[xc] = data[xc];

    for(xc=halfk; xc<dsize-halfk; xc++) {
        long xk;
        double temp = 0;
        for (xk=0; xk<ksize; xk++)
            temp += kernel[xk]*data[xc-halfk+xk];
        convolved[xc] = temp;
    }

    for(xc=dsize-halfk; xc<dsize; xc++)
        convolved[xc] = data[xc];
}

```

## 10.4 Element-wise API

The element-wise in-place API is a family of macros and functions designed to get and set elements of arrays which might be byteswapped, misaligned, discontinuous, or of a different type. You can obtain `PyArrayObjects` for these misbehaved arrays from the high-level API by specifying fewer requirements (perhaps just 0, rather than `NUM_C_ARRAY`). In this way, you can avoid the creation of temporaries at a cost of accessing your array with these macros and functions and a significant performance penalty. Make no mistake, if you have the memory, the high level API is the fastest. The whole point of this API is to support cases where the creation of temporaries exhausts either the physical or virtual address space. Exhausting physical memory will result in thrashing, while exhausting the virtual address space will result in program exception and failure. This API supports avoiding the creation of the temporaries, and thus avoids exhausting physical and virtual memory, possibly improving net performance or even enabling program success where simpler methods would just fail.

### 10.4.1 Element-wise functions

The single element macros each access one element of an array at a time, and specify the array type in two places: as part of the `PyArrayObject` type descriptor, and as “type”. The former defines what the array is, and the latter is required to produce correct code from the macro. They should *match*. When you pass “type” into one of these macros, you are defining the kind of array the code can operate on. It is an error to pass a non-matching array to one of these macros. One last piece of advice: call these macros carefully, because the resulting expansions and error messages are a *\*obscene\**. Note: the type parameter for a macro is one of the Numarray Numeric Data Types, not a NumarrayType enumeration value.

#### Pointer based single element macros

**NA\_GETPa** (*PyArrayObject\**, *type*, *char\**)  
aligning

**NA\_GETPb** (*PyArrayObject\**, *type*, *char\**)  
byteswapping

**NA\_GETPf** (*PyArrayObject\**, *type*, *char\**)  
fast (well-behaved)

**NA\_GETP** (*PyArrayObject\**, *type*, *char\**)  
testing: any of above

**NA\_SETPa** (*PyArrayObject\**, *type*, *char\**, *v*)

**NA\_SETPb** (*PyArrayObject\**, *type*, *char\**, *v*)

**NA\_SETPf** (*PyArrayObject\**, *type*, *char\**, *v*)

**NA\_SETP** (*PyArrayObject\**, *type*, *char\**, *v*)

#### One index single element macros

**NA\_GET1a** (*PyArrayObject\**, *type*, *i*)

**NA\_GET1b** (*PyArrayObject\**, *type*, *i*)

**NA\_GET1f** (*PyArrayObject\**, *type*, *i*)

**NA\_GET1** (*PyArrayObject\**, *type*, *i*)

**NA\_SET1a** (*PyArrayObject\**, *type*, *i*, *v*)

**NA\_SET1b** (*PyArrayObject\**, *type*, *i*, *v*)



```
NA_SET1f(PyArrayObject*, type, i, v)
```

```
NA_SET1(PyArrayObject*, type, i, v)
```

Two index single element macros

```
NA_GET2a(PyArrayObject*, type, i, j)
```

```
NA_GET2b(PyArrayObject*, type, i, j)
```

```
NA_GET2f(PyArrayObject*, type, i, j)
```

```
NA_GET2(PyArrayObject*, type, i, j)
```

```
NA_SET2a(PyArrayObject*, type, i, j, v)
```

```
NA_SET2b(PyArrayObject*, type, i, j, v)
```

```
NA_SET2f(PyArrayObject*, type, i, j, v)
```

```
NA_SET2(PyArrayObject*, type, i, j, v)
```

One and Two Index, Offset, Float64/Complex64/Int64 functions

The Int64/Float64/Complex64 functions require a function call to access a single element of an array, making them slower than the single element macros. They have two advantages:

1. They're function calls, so they're a little more robust.
2. They can handle *any* input array type and behavior properties.

While these functions have no error return status, they *can* alter the Python error state, so well written extensions should call `PyErr_Occurred()` to determine if an error occurred and report it. It's reasonable to do this check once at the end of an extension function, rather than on a per-element basis.

```
void NA_get_offset(PyArrayObject *, int N, ...)
```

`NA_get_offset` computes the offset into an array object given a variable number of indices. It is not especially robust, and it is considered an error to pass it more indices than the array has, or indices which are negative or out of range.

```
Float64 NA_get_Float64(PyArrayObject *, long offset)
```

```
void NA_set_Float64(PyArrayObject *, long offset, Float64 v)
```

```
Float64 NA_get1_Float64(PyArrayObject *, int i)
```

```
void NA_set1_Float64(PyArrayObject *, int i, Float64 v)
```

```
Float64 NA_get2_Float64(PyArrayObject *, int i, int j)
```

```
void NA_set2_Float64(PyArrayObject *, int i, int j, Float64 v)
```

```
Int64 NA_get_Int64(PyArrayObject *, long offset)
```

```
void NA_set_Int64(PyArrayObject *, long offset, Int64 v)
```

```
Int64 NA_get1_Int64(PyArrayObject *, int i)
```

```
void NA_set1_Int64(PyArrayObject *, int i, Int64 v)
```

```
Int64 NA_get2_Int64(PyArrayObject *, int i, int j)
```

```
void NA_set2_Int64(PyArrayObject *, int i, int j, Int64 v)
```

```
Complex64 NA_get_Complex64(PyArrayObject *, long offset)
```

```

void NA_set_Complex64(PyArrayObject *, long offset, Complex64 v)
Complex64 NA_get1_Complex64(PyArrayObject *, int i)
void NA_set1_Complex64(PyArrayObject *, int i, Complex64 v)
Complex64 NA_get2_Complex64(PyArrayObject *, int i, int j)
void NA_set2_Complex64(PyArrayObject *, int i, int j, Complex64 v)

```

## 10.4.2 Example

The `convolve1d` wrapper function corresponding to section 10.3.3 using the element-wise API could look like:<sup>2</sup>

```

static PyObject *
Py_Convolve1d(PyObject *obj, PyObject *args)
{
    PyObject *okernel, *odata, *oconvolved=Py_None;
    PyArrayObject *kernel, *data, *convolved;

    if (!PyArg_ParseTuple(args, "OO|O", &okernel, &odata, &oconvolved))
        return PyErr_Format(_Error,
                             "Convolve1d: Invalid parameters.");

    kernel = NA_InputArray(okernel, tAny, 0);
    data = NA_InputArray(odata, tAny, 0);

```

For the kernel and data arrays, numarrays of any type are accepted without conversion. Thus there is no copy of the data made except for lists or tuples. All types, byteswapping, misalignment, and discontiguity must be handled by `Convolve1d`. This can be done easily using the get/set functions. Macros, while faster than the functions, can only handle a single type.

```

convolved = NA_OptionalOutputArray(oconvolved, tFloat64, 0, data);

```

Also for the output array we accept any variety of type `tFloat` without conversion. No copy is made except for non-`tFloat`. Non-numarray sequences are not permitted as output arrays. Byteswaping, misalignment, and discontiguity must be handled by `Convolve1d`. If the Pythoncaller did not specify the `oconvolved` array, it initially retains the value `Py_None`. In that case, *convolved* is cloned from the array *data* using the specified type. It is important to clone from *data* and not *odata*, since the latter may be an ordinary Pythonsequence which was converted into numarray *data*.

<sup>2</sup>This function is also available as an example in the source distribution.

```

    if (!kernel || !data || !convolved)
        return NULL;

    if ((kernel->nd != 1) || (data->nd != 1))
        return PyErr_Format(_Error,
                             "Convolved: arrays must have exactly 1
                             dimension.");

    if (!NA_ShapeEqual(data, convolved))
        return PyErr_Format(_Error,
                             "Convolved: data and output arrays must have identical length.");
    if (!NA_ShapeLessThan(kernel, data))
        return PyErr_Format(_Error,
                             "Convolved: kernel must be smaller than data in both dimensions");

    if (Convolved(kernel, data, convolved) < 0) /* Error? */
        return NULL;
    else {
        Py_XDECREF(kernel);
        Py_XDECREF(data);
        return NA_ReturnOutput(oconvolved, convolved);
    }
}

```

This function is very similar to the high-level API wrapper, the notable difference is that we ask for the unconverted arrays *kernel* and *data* and *convolved*. This requires some attention in their usage. The function that does the actual convolution in the example has to use `NA_get*` to read and `NA_set*` to set an element of these arrays, instead of using straight array notation. These functions perform any necessary type conversion, byteswapping, and alignment.

```

static int
Convolve1d(PyArrayObject *kernel, PyArrayObject *data, PyArrayObject *convolved)
{
    int xc, xk;
    int ksize = kernel->dimensions[0];
    int halfk = ksize / 2;
    int dsize = data->dimensions[0];

    for(xc=0; xc<halfk; xc++)
        NA_set1_Float64(convolved, xc, NA_get1_Float64(data, xc));

    for(xc=dsize-halfk; xc<dsize; xc++)
        NA_set1_Float64(convolved, xc, NA_get1_Float64(data, xc));

    for(xc=halfk; xc<dsize-halfk; xc++) {
        Float64 temp = 0;
        for (xk=0; xk<ksize; xk++) {
            int i = xc - halfk + xk;
            temp += NA_get1_Float64(kernel, xk) *
                    NA_get1_Float64(data, i);
        }
        NA_set1_Float64(convolved, xc, temp);
    }
    if (PyErr_Occurred())
        return -1;
    else
        return 0;
}

```

## 10.5 One-dimensional API

The 1D in-place API is a set of functions for getting/setting elements from the innermost dimension of an array. These functions improve speed by moving type switches, “behavior tests”, and function calls out of the per-element loop. The functions get/set a series of consecutive array elements to/from arrays of `Int64`, `Float64`, or `Complex64`. These functions are (even) more intrusive than the single element functions, but have better performance in many cases. They can operate on arrays of any type, with the exception of the `Complex64` functions, which only handle `Complex64`. The functions return 0 on success and -1 on failure, with the Python error state already set. To be used profitably, the 1D API requires either a large single dimension which can be processed in blocks or a multi-dimensional array such as an image. In the latter case, the 1D API is suitable for processing one (or more) scanlines at a time rather than the entire image at once. See the source distribution `Examples/convolve/one_dimensionalmodule.c` for an example of usage.

long **NA\_get\_offset** (*PyArrayObject* \*, *int* *N*, ...)

This function applies a (variable length) set of *N* indices to an array and returns a byte offset into the array.

int **NA\_get1D\_Int64** (*PyArrayObject* \*, long *offset*, int *cnt*, *Int64* \**out*)

int **NA\_set1D\_Int64** (*PyArrayObject* \*, long *offset*, int *cnt*, *Int64* \**in*)

int **NA\_get1D\_Float64** (*PyArrayObject* \*, long *offset*, int *cnt*, *Float64* \**out*)

int **NA\_set1D\_Float64** (*PyArrayObject* \*, long *offset*, int *cnt*, *Float64* \**in*)

int **NA\_get1D\_Complex64** (*PyArrayObject* \*, long *offset*, int *cnt*, *Complex64* \**out*)

int **NA\_set1D\_Complex64** (*PyArrayObject* \*, long *offset*, int *cnt*, *Complex64* \**in*)

## 10.6 Numeric emulation API

These notes describe the Numeric compatability functions which enable numarray to utilize a subset of the extensions written for Numeric (NumPy). Not all Numeric C-API features and therefore not all Numeric extensions are currently supported. Users should be able to utilize suitable extensions written for Numeric within the numarray environment by:

1. Writing a numarray setup.py file.
2. Scanning the extension C-code for all instances of array creation and return and making corrections as needed and specified below.
3. Re-compiling the Numeric C-extension for numarray.

Numarray's compatability with Numeric consists of 3 things:

1. A replacement header file, "arrayobject.h" which supplies emulation functions and macros for numarray just as the original arrayobject.h supplies the C-API for Numeric.
2. Layout and naming of the fundamental numarray C-type, `PyArrayObject`, in Numeric compatible way.
3. A set of "emulation" functions. These functions have the same names and parameters as the original Numeric functions, but operate on numarrays. The emulation functions are also incomplete; features not currently supported should result in compile time warnings.

`PyArrayObject` was discussed in an earlier section. The header file, "arrayobject.h" is now fairly minimal, mainly just including `libnumarray.h`, so needs little discussion. The following section will discuss the Numeric compatible functions.

### 10.6.1 Emulation Functions

The basic use of numarrays by Numeric extensions is achieved in the extension function's wrapper code by:

1. Ensuring creation of array objects by calls to emulation functions.
2. DECREFing each array and/or calling `PyArray_Return`.

Unlike prior versions of numarray, this version *\*does\** support access to array objects straight out of `PyArg_ParseTuple`. This is a consequence of a change to the underlying object model, where a class instance has been replaced by `PyArrayObject`. Nevertheless, the "right" way to access arrays is either via the high level interface or via emulated Numeric factory functions. That way, access to other python sequences is supported as well.

The creation of array objects is illustrated by the following of wrapper code for a 2D convolution function:

```
static PyObject *
Py_Convolve2d(PyObject *obj, PyObject *args)
{
    PyObject *okernel, *odata, *oconvolved=Py_None;
    PyArrayObject *kernel, *data, *convolved;

    if (!PyArg_ParseTuple(args, "OO|O", &okernel, &odata, &oconvolved))
        return PyErr_Format(_Error,
                           "Convolve2d: Invalid parameters.");
}
```

The first step was simply to get object pointers to the ndarray parameters to the convolution function: `okernel`, `odata`, and `oconvolved`. `Oconvolved` is an optional output parameter, specified with a default value of `Py_None` which is used when only 2 parameters are supplied at the python level. Each of the “o” parameters should be thought of as an arbitrary sequence object, not necessarily an array.

The next step is to call emulation functions which convert sequence objects into `PyArrayObjects`. In a Numeric extension, these calls map tuples and lists onto Numeric arrays and assert their dimensionality as 2D. The ndarray emulation functions first map tuples, lists, and misbehaved ndarrays onto well-behaved ndarrays. Thus, calls to the emulation factory functions transparently use the ndarray high level interface and provide visibility only to aligned and non-byteswapped array objects.

```
kernel = (PyArrayObject *) PyArray_ContiguousFromObject(
    okernel, tFloat64, 2, 2);
data = (PyArrayObject *) PyArray_ContiguousFromObject(
    odata, tFloat64, 2, 2);

if (!kernel || !data) return NULL;
```

Extra processing is required to handle the output array *convolved*, cloning it from *data* if it was not specified. Code should be supplied, but is not, to verify that *convolved* and *data* have the same shape.

```
if (convolved == Py_None)
    convolved = (PyArrayObject *) PyArray_FromDims(
        data->nd, data->dimensions, tFloat64);
else
    convolved = (PyArrayObject *) PyArray_ContiguousFromObject(
        oconvolved, tFloat64, 2, 2);
if (!convolved) return NULL;
```

After converting all of the input parameters into `PyArrayObjects`, the actual convolution is performed by a separate function. This could just as well be done inline:

```
Convolve2d(kernel, data, convolved);
```

After processing the arrays, they should be `DECREF`'ed or returned using `PyArray_Return`. It is generally not possible to directly return a ndarray object using `Py_BuildValue` because the shadowing of mis-behaved arrays needs to be undone. Calling `PyArray_Return` destroys any temporary and passes the ndarray back to Python.

```
Py_DECREF(kernel);
Py_DECREF(data);
if (convolved != Py_None) {
    Py_DECREF(convolved);
    Py_INCREF(Py_None);
    return Py_None;
} else
    return PyArray_Return(convolved);
}
```

Byteswapped or misaligned arrays are handled by a process of shadowing which works like this:

1. When a “misbehaved” ndarray is accessed via the Numeric emulation functions, first a well-behaved tempo-

rary copy (shadow) is created by `NA_IoArray`.

2. Operations performed by the extension function modify the data buffer belonging to the shadow.
3. On extension function exit, the shadow array is copied back onto the original and the shadow is freed.

All of this is transparent to the user; if the original array is well-behaved, it works much like it always did; if not, what would have failed altogether works at the cost of extra temporary storage. Users which cannot afford the cost of shadowing need to use `numarray`'s native elementwise or 1D APIs.

## 10.6.2 Numeric Compatible Functions

The following functions are currently implemented:

`PyArrayObject*` **PyArray\_FromDims**(*int* nd, *int* \*dims, *int* type)

This function will allocate a new `numarray`.

An array created with `PyArray_FromDims` can be used as a temporary or returned using `PyArray_Return`.

Used as a temporary, calling `Py_DECREF` deallocates it.

`PyObject*` **PyArray\_FromDimsAndData**(*int* nd, *int* \*dims, *int* type, *char* \*data)

This function will allocate a `numarray` of the specified shape and type, and its contents will be copied from the specified data pointer.

`PyObject*` **PyArray\_ContiguousFromObject**(*PyObject* \*op, *int* type, *int* min\_dim, *int* max\_dim)

Returns an emulation object for a contiguous `numarray` of 'type' created from the sequence object 'op'. If 'op' is a contiguous, aligned, non-byteswapped `numarray`, then the emulation object refers to it directly. Otherwise a well-behaved `numarray` will be created from 'op' and the emulation object will refer to it. `min_dim` and `max_dim` bound the expected rank as in Numeric. `min_dim==max_dim` specifies an exact rank. `min_dim==max_dim==0` specifies *any* rank.

`PyObject*` **PyArray\_CopyFromObject**(*PyObject* \*op, *int* type, *int* min\_dim, *int* max\_dim)

Returns a contiguous array, similar to `PyArray_FromContiguousObject`, but always returning an emulation object referring to a new `numarray` copied from the original sequence.

`PyObject*` **PyArray\_FromObject**(*PyObject* \*op, *int* type, *int* min\_dim, *int* max\_dim)

Returns an emulation object based on 'op', possibly discontinuous. The strides array must be used to access elements of the emulation object.

If 'op' is a byteswapped or misaligned `numarray`, `FromObject` creates a temporary copy and the emulation object refers to it.

If 'op' is a nonswapped, aligned `numarray`, the emulation object refers to it.

If 'op' is some other sequence, it is converted to a `numarray` and the emulation object refers to that.

`PyObject*` **PyArray\_Return**(*PyArrayObject* \*apr)

Returns emulation object 'apr' to python. The emulation object itself is destructed. The `numarray` it refers to (base) is returned as the result of the function.

An additional check is (or eventually will be) performed to guarantee that rank-0 arrays are converted to appropriate python scalars.

`PyArray_Return` has no net effect on the reference count of the underlying `numarray`.

*int* **PyArray\_As1D**(*PyObject* \*\*op, *char* \*\*ptr, *int* \*d1, *int* typecode)

Copied from Numeric verbatim.

*int* **PyArray\_As2D**(*PyObject* \*\*op, *char* \*\*\*ptr, *int* \*d1, *int* \*d2, *int* typecode)

Copied from Numeric verbatim.

*int* **PyArray\_Free**(*PyObject* \*op, *char* \*ptr)

Copied from Numeric verbatim. **Note:** This means including bugs and all!

```

int PyArray_Check(PyObject *op)
    This function returns 1 if op is a PyArrayObject.

int PyArray_Size(PyObject *op)
    This function returns the total element count of the array.

int PyArray_NBYTES(PyArrayObject *op)
    This function returns the total size in bytes of the array, and assumes that bytestride == itemsize, so that the size
    is product(shape)*itemsize.

PyObject* PyArray_Copy(PyArrayObject *op)
    This function returns a copy of the array 'op'.

int PyArray_CanCastSafely(PyArrayObject *op, int type)
    This function returns 1 IFF the array 'op' can be safely cast to 'type', otherwise it returns 0.

PyArrayObject* PyArray_Cast(PyArrayObject *op, int type)
    This function casts the array 'op' into an equivalent array of type 'type'.

PyArray_Descr* PyArray_DescrFromType(int type)
    This function returns a pointer to the array descriptor for 'type'. The numarray version of PyArray_Descr is
    incomplete and does not support casting, getitem, setitem, one, or zero.

int PyArray_isArray(PyObject *o)(T)
    This macro is designed to fail safe and return 0 when numarray is not installed at all. When numarray is installed,
    it returns 1 iff object 'o' is a numarray, and 0 otherwise. This macro facilitates the optional use of numarray
    within an extension.

```

### 10.6.3 Unsupported Numeric Features

- PyArrayError
- PyArray\_ObjectType()
- PyArray\_Reshape()
- PyArray\_SetStringFunction()
- PyArray\_SetNumericOps()
- PyArray\_Take()
- UFunc API

## 10.7 New numarray functions

There are several functions to create numarrays at the C level:

```

static PyObject* NA_NewArray(void *buffer, NumarrayType type, int ndim, ...)
    Create a new numarray from C data buffer. The new array has type type, ndim dimensions, and the length
    of each dimension must be given as the remaining (variable length) list of int parameters. Data is copied from
    buffer into the memory object of the new array.

static PyObject* NA_vNewArray(void *buffer, NumarrayType type, int ndim, maybelong *shape)
    Create a new numarray from C data buffer. The new array has type type, ndim dimensions, and the length
    of each dimension must be given as an array of long pointed to by shape. Data is copied from buffer into the
    memory object of the new array.

```



```
static PyObject* NA_NewAll(int ndim, maybelong *shape, NumarrayType type, void *buffer, maybelong
                           byteoffset, maybelong bytearray, int aligned, int writable)
    numarray from C data buffer. The new array has type type, ndim dimensions, and the length of each dimension
    must be given in shape[ndim]. byteoffset, bytestride specify the data-positions in the C array to use. byte-
order and aligned specify the corresponding parameters. byteorder takes one of the values NUM_BIG_ENDIAN
    or NUM_LITTLE_ENDIAN. writable defines whether the buffer object associated with the resulting array is
    readonly or writable. Data is copied from buffer into the memory object of the new array.
```

```
static PyObject* NA_NewAllStrides(int ndim, maybelong *shape, maybelong *strides, NumarrayType
                                   type, void *buffer, maybelong byteoffset, maybelong bytearray, int
                                   aligned, int writable)
    numarray from C data buffer. The new array has type type, ndim dimensions, and the length of each dimension
    must be given in shape[ndim]. The byte offset between successive indices of a dimension is specified by
    strides[ndim]. byteoffset specifies the offset from the base of the buffer to the zeroth element of the array. byte-
order and aligned specify the corresponding parameters. byteorder takes one of the values NUM_BIG_ENDIAN
    or NUM_LITTLE_ENDIAN. writable defines whether the buffer object associated with the resulting array is
    readonly or writable. Data is copied from buffer into the memory object of the new array.
```

```
int NA_ShapeEqual(PyArrayObject*a,PyArrayObject*b)
    This function compares the shapes of two arrays, and returns 1 if they are the same, 0 otherwise.
```

```
int NA_ShapeLessThan(PyArrayObject*a,PyArrayObject*b)
    This function compares the shapes of two arrays, and returns 1 if each dimension of 'a' is less than the corre-
    sponding dimension of 'b', 0 otherwise.
```

```
int NA_ByteOrder( )
    This function returns the system byte order, either NUM_LITTLE_ENDIAN or NUM_BIG_ENDIAN.
```

```
Bool NA_Ieeespecial32(Float32 *f, Int32 *mask)
    This function returns 1 IFF Float32 value '*f' matches any of the IEEE special value criteria specified by
    '*mask'. See ieeespecial.h for the mask bit values which can be or'ed together to specify mask.
```

```
Bool NA_Ieeespecial64(Float64*f,Int32*)
    This function returns 1 IFF Float64 value '*f' matches any of the IEEE special value criteria specified by
    '*mask'. See ieeespecial.h for the mask bit values which can be or'ed together to specify mask.
```

```
PyArrayObject * NA_updateDataPtr(PyArrayObject *)
    This function updates the values derived from the "_data" buffer, namely the data pointer and buffer WRITABLE
    flag bit. This needs to be called upon entering or re-entering C-code from Python, since it is possible for buffer
    objects to move their data buffers as a result of executing arbitrary Python and hence arbitrary C-code.
```

```
char* NA_typeNoToName(int)
    NA_typeNoToName translates a NumarrayType into a character string which can be used to display it: e.g.
    tInt32 converts to the string "Int32"
```

```
PyObject* NA_typeNoToTypeObject(int)
    This function converts a NumarrayType C type code into the NumericType object which implements and repre-
    sents it more fully. tInt32 converts to the type object numarray.Int32.
```

```
int NA_typeObjectToTypeNo(PyObject*)
    This function converts a numarray type object (e.g. numarray.Int32) into the corresponding NumarrayType (e.g.
    tInt32) C type code.
```

```
PyObject* NA_intTupleFromMaybeLongs(int,maybelong*)
    This function creates a tuple of Python ints from an array of C maybelong integers.
```

```
long NA_maybeLongsFromIntTuple(int,maybelong*,PyObject*)
    This function fills an array of C long integers with the converted values from a tuple of Python ints. It returns
    the number of conversions, or -1 for error.
```

```
long NA_isIntegerSequence(PyObject*)
```

This function returns 1 iff the single parameter is a sequence of Python integers, and 0 otherwise.

`PyObject* NA_setArrayFromSequence(PyArrayObject*,PyObject*)`

This function copies the elementwise from a sequence object to a ndarray.

`int NA_maxType(PyObject*)`

This function returns an integer code corresponding to the highest kind of Python numeric object in a sequence. INT(0) LONG(1) FLOAT(2) COMPLEX(3). On error -1 is returned.

`PyObject* NA_getPythonScalar(PyArrayObject *a, long offset)`

This function returns the Python object corresponding to the single element of the array 'a' at the given byte offset.

`int NA_setFromPythonScalar(PyArrayObject *a, long offset, PyObject*value)`

This function sets the single element of the array 'a' at the given byte offset to 'value'.

`int NA_NDArrayCheck(PyObject*o)`

This function returns 1 iff the 'o' is an instance of NDArray or an instance of a subclass of NDArray, and 0 otherwise.

`int NA_NumArrayCheck(PyObject*)`

This function returns 1 iff the 'o' is an instance of NumArray or an instance of a subclass of NumArray, and 0 otherwise.

`int NA_ComplexArrayCheck(PyObject*)`

This function returns 1 iff the 'o' is an instance of ComplexArray or an instance of a subclass of ComplexArray, and 0 otherwise.

`int NA_elements(PyArrayObject*)`

This function returns the total count of elements in an array, essentially the product of the elements of the array's shape.

`PyArrayObject * NA_copy(PyArrayObject*)`

This function returns a copy of the given array.

`int NA_copyArray(PyArrayObject*to, const PyArrayObject *from)`

This function returns a copies one array onto another; used in f2py.



## **Part II**

# **Extension modules**



# Convolution

This package provides functions for one- and two-dimensional convolutions and correlations of `numarrays`.

## 11.1 Convolution functions

**boxcar** (*data*, *boxshape*, *output*=None, *mode*='nearest', *cval*=0.0)

`boxcar` computes a 1D or 2D boxcar filter on every 1D or 2D subarray of *data*. *boxshape* is a tuple of integers specifying the dimensions of the filter, e.g. ( 3 , 3 ). If *output* is specified, it should be the same shape as *data* and the result will be stored in it. In that case None will be returned.

Supported *modes* include:

**nearest**:Elements beyond boundary come from nearest edge pixel.

**wrap**:Elements beyond boundary come from the opposite array edge.

**reflect**:Elements beyond boundary come from reflection on same array edge.

**constant**:Elements beyond boundary are set to 'cval', an optional numerical parameter; the default value is 0 . 0 .

**convolve** (*data*, *kernel*, *mode*=FULL)

Returns the discrete, linear convolution of 1-D sequences *data* and *kernel*. *mode* can be VALID, SAME, or FULL to specify the size of the resulting sequence. See section 11.2.

**convolve2d** (*data*, *kernel*, *output*=None, *fft*=0, *mode*='nearest', *cval*=0.0)

Return the 2-dimensional convolution of *Data* and *kernel*. If *output* is not None, the result is stored in *output* and None is returned. *fft* is used to switch between FFT based convolution and the naive algorithm, defaulting to naive. *fft* mode becomes beneficial as the size of the kernel grows; for small kernels, the naive algorithm is more efficient. *modes* are identical to those of `boxcar`.

**correlate** (*data*, *kernel*, *mode*=FULL)

Return the cross-correlation of *data* and *kernel*. *mode* can be VALID, SAME, or FULL to specify the size of the resulting sequence. See section 11.2.

**correlate2d** (*data*, *kernel*, *output*=None, *fft*=0, *mode*='nearest', *cval*=0.0)

Return the 2-dimensional correlation of *Data* and *kernel*. If *output* is not None, the result is stored in *output* and None is returned. *fft* is used to switch between FFT based correlation and the naive algorithm, defaulting to naive. *fft* mode becomes beneficial as the size of the kernel grows; for small kernels, the naive algorithm is more efficient. *modes* are identical to those of `boxcar`.

**Note:** `cross_correlate` is deprecated and should not be used.

## 11.2 Global constants

These constants specify what part of the result the `convolve` and `correlate` functions of this module return. Each of the following examples assumes that the following code has been executed:

```
import numpy.convolve as conv
```

#### **FULL**

Return the full convolution or correlation of two arrays.

```
>>> conv.cross_correlate(numpy.arange(8), [1, 2, 3], mode=conv.FULL)
array([ 0,  3,  8, 14, 20, 26, 32, 38, 20,  7])
```

#### **PASS**

Correlate the arrays without padding the data.

```
>>> conv.cross_correlate(numpy.arange(8), [1, 2, 3], mode=conv.PASS)
array([ 0,  8, 14, 20, 26, 32, 38,  7])
```

#### **SAME**

Return the part of the convolution or correlation of two arrays that corresponds to an array of the same shape as the input data.

```
>>> conv.cross_correlate(numpy.arange(8), [1, 2, 3], mode=conv.SAME)
array([ 3,  8, 14, 20, 26, 32, 38, 20])
```

#### **VALID**

Return the valid part of the convolution or correlation of two arrays.

```
>>> conv.cross_correlate(numpy.arange(8), [1, 2, 3], mode=conv.VALID)
array([ 8, 14, 20, 26, 32, 38])
```

# Fast-Fourier-Transform

This package provides functions for one- and two-dimensional Fast-Fourier-Transforms (FFT) and inverse FFTs.

The `numarray.fft` 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.

## 12.1 Installation

The default installation uses the provided `numarray.fft.fftpack` C implementation of these routines and this works without any further interaction.

### 12.1.1 Installation using FFTPACK

On some platforms, precompiled optimized versions of the FFTPACK libraries are preinstalled on the operating system, and the setup procedure needs to be modified to force the `numarray.fft` module to be linked against those rather than the builtin replacement functions.

## 12.2 FFT Python Interface

The Python user imports the `numarray.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 `len(data)`, then *data* will be zero-padded to make up the difference. If *n* is smaller than `len(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.

Some examples are:



```

>>> a = array([1., 0., 1., 0., 1., 0., 1., 0.]) + 10
>>> b = array([0., 1., 0., 1., 0., 1., 0., 1.]) + 10
>>> c = array([0., 1., 0., 0., 0., 1., 0., 0.]) + 10
>>> print numarray.fft.fft(a).real
[ 84.  0.  0.  0.  4.  0.  0.  0.]
>>> print numarray.fft.fft(b).real
[ 84.  0.  0.  0. -4.  0.  0.  0.]
>>> print numarray.fft.fft(c).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 numarray.fft.real_fft(x)
[ -5.82867088e-16 +0.00000000e+00j  1.50000000e+01 -3.08862614e-15j
  7.13643755e-16 -1.04457106e-15j  1.13047653e-15 -3.23843935e-15j
 -1.52158521e-15 +1.14787259e-15j  3.60822483e-16 +3.60555504e-16j
  1.34237661e-15 +2.05127011e-15j  1.98981960e-16 -1.02472357e-15j
  1.55899290e-15 -9.94619821e-16j -1.05417678e-15 -2.33364171e-17j
 -2.08166817e-16 +1.00955541e-15j -1.34094426e-15 +8.88633386e-16j
  5.67513742e-16 -2.24823896e-15j  2.13735778e-15 -5.68448962e-16j
 -9.55398954e-16 +7.76890265e-16j -1.05471187e-15 +0.00000000e+00j]

```

**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* / .

## 12.3 fftpack Python Interface

The interface to the FFTPACK library is performed via the `fftpack` 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:

**cfft1**(*i*)

**cfft1f**(*data*, *savearea*)

**cfft1b**(*data*, *savearea*)

**rfft1**(*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 initialization 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.

```
>>> import numpy.fft.fftpack as fftpack
>>> x = cos(arange(30.0)/30.0*2*pi)
>>> w = fftpack.rfffti(30)
>>> f = fftpack.rfftf(x, w)
>>> print f[0:5]
[ -5.68989300e-16 +0.00000000e+00j   1.50000000e+01 -3.08862614e-15j
   6.86516117e-16 -1.00588467e-15j   1.12688689e-15 -3.19983494e-15j
  -1.52158521e-15 +1.14787259e-15j]
```



# Linear Algebra

The `numarray.linear_algebra` module provides a simple interface to some commonly used linear algebra routines.

The `numarray.linear_algebra` module provides a simple high-level interface to some common linear algebra problems. It uses either the LAPACK Fortran library or the compatible `numarray.linear_algebra.lapack_lite` C library shipped with `numarray`.

## 13.1 Installation

The default installation uses the provided `numarray.linear_algebra.lapack_lite` implementation of these routines and this works without any further interaction.

Nevertheless if LAPACK is installed already or you are concerned about the performance of these routines you should consider installing `numarray.linear_algebra` to take advantage of the real LAPACK library. See the next section for instructions.

### 13.1.1 Installation using LAPACK

On some platforms, precompiled optimized versions of the LAPACK and BLAS libraries are preinstalled on the operating system, and the setup procedure needs to be modified to force the `lapack_lite` module to be linked against those rather than the builtin replacement functions.

Edit `Packages/LinearAlgebra2/setup.py` and edit the variables `sourcelist`, `lapack_dirs`, and `lapack_libs`.

In `sourcelist` you should remove all sourcefiles besides `lapack_litemodule.c`. In `lapack_libs` you have to specified all libraries needed on your system to successfully link against LAPACK. Often this includes 'lapack' and 'blas'. If you need to specify any non-standard directories to find these libraries, you can specify them in `lapack_dirs`.

**Note:** A frequent request is that somehow the maintainers of Numerical Python invent a procedure which will automatically find and use the *best* available versions of these libraries. We welcome any patches that provide the functionality in a simple, platform independent, and reliable way. The [scipy](#) project has done some work to provide such functionality, but is probably not mature enough for use by `numarray` yet.

## 13.2 Python Interface

All examples in this section assume that you performed a

```
from numpy import *
import numpy.linalg as la
```

### **cholesky\_decomposition(a)**

This function returns a lower triangular matrix L which, when multiplied by its transpose yields the original matrix a. a must be square, Hermitian and positive definite. L is often referred to as the Cholesky lower-triangular square-root of a.

### **determinant(a)**

This function returns the determinant of the square matrix a.

### **eigenvalues(a)**

This function returns the eigenvalues of the square matrix a.

```
>>> a = array([[1., 0., 0., 0.], [0., 2., 0., 0.01], [0., 0., 5., 0.], [0., 0.01, 0., 2.5]])
>>> print a
[[ 1.  0.  0.  0. ]
 [ 0.  2.  0.  0.01]
 [ 0.  0.  5.  0. ]
 [ 0.  0.01 0.  2.5 ]]
>>> la.eigenvalues(a)
array([ 2.50019992,  1.99980008,  1.          ,  5.          ])
```

### **eigenvectors(a)**

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

```
>>> a = array([[1., 0., 0., 0.], [0., 2., 0., 0.01], [0., 0., 5., 0.], [0., 0.01, 0., 2.5]])
>>> print a
[[ 1.  0.  0.  0. ]
 [ 0.  2.  0.  0.01]
 [ 0.  0.  5.  0. ]
 [ 0.  0.01 0.  2.5 ]]
>>> eval, evec = la.eigenvectors(a)
>>> print eval
[ 2.50019992  1.99980008  1.          5.          ]
>>> print transpose(evec)
[[ 0.          0.          1.          0.          ]
 [ 0.01998801  0.99980022  0.          0.          ]
 [ 0.          0.          0.          1.          ]
 [ 0.99980022 -0.01998801  0.          0.          ]]
```

### **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.

### **Heigenvalues(a)**

returns the (real positive) eigenvalues of the square, Hermitian positive definite matrix a.

### **Heigenvectors(a)**

returns both the (real positive) eigenvalues and the eigenvectors of a square, Hermitian positive definite matrix a. The eigenvectors are returned in an (orthonormal) two-dimensional matrix.

### **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 = la.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  $a a^{-1} - \text{identity}(5)$ :

```
>>> print "Inversion error:", maximum.reduce(fabs(ravel(dot(a, inv_a)-identity(5))))
Inversion error: 8.18789480661e-16
```

#### **`linear_least_squares(a, b, rcond=1e-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.

#### **`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*.

#### **`singular_value_decomposition(a, full_matrices=0)`**

This function returns three arrays *V*, *S*, and *WT* whose matrix product is the original matrix *a*. *V* and *WT* 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.



# Masked Arrays

Masked arrays are arrays that may have missing or invalid entries. Module MA provides a nearly work-alike replacement for Numeric that supports data arrays with masks.

## Required Packages

MA uses `numarray` and the optional package `Properties`.

### 14.1 What is a masked array?

Masked arrays are arrays that may have missing or invalid entries. Module MA provides a work-alike replacement for `numarray` that supports data arrays with masks. A mask is either `None` or an array of ones and zeros, that determines for each element of the masked array whether or not it contains an invalid entry. The package assures that invalid entries are not. A particular element is said to be masked (invalid) if the mask is not `None` and the corresponding element of the mask is 1; otherwise it is unmasked (valid).

This package was written by Paul F. Dubois at Lawrence Livermore National Laboratory. Please see the legal notice in the software and section “License and disclaimer for packages MA, RNG, Properties”.

### 14.2 Installing and using MA

MA is one of the optional Packages and installing it requires a separate step as explained in the `numarray` README. To install just the MA package using Distutils, in the MA top directory enter:

```
python setup.py install
```

Use MA as a replacement for Numeric:

```
from MA import *
>>> x = array([1, 2, 3])
```

To create an array with the second element invalid, we would do:



```
>>> y = array([1, 2, 3], mask = [0, 1, 0])
```

To create a masked array where all values “near”  $1.e20$  are invalid, we can do:

```
>>> z = masked_values([1.0, 1.e20, 3.0, 4.0], 1.e20)
```

For a complete discussion of creation methods for masked arrays please see section ?? “Constructing masked arrays”.

The `numarray` module is an attribute in `MA`, so to execute a method `foo` from `Numeric`, you can reference it as `numarray.foo`.

Usually people use both `MA` and `Numeric` this way, but of course you can always fully-qualify the names:

```
>>> import MA
>>> x = MA.array([1, 2, 3])
```

The principal feature of module `MA` is class `MaskedArray`, the class whose instances are returned by the array constructors and most functions in module `MA`. We will discuss this class first, and later cover the attributes and functions in module `MA`. For now suffice it to say that among the attributes of the module are the constants from module `numarray` including those for declaring typecodes, `NewAxis`, and the mathematical constants such as `pi` and `e`. An additional typecode, `MaskType`, is the typecode used for masks.

## 14.3 Class `MaskedArray`

In Module `MA`, an array is an instance of class `MaskedArray`, which is defined in the module `MA`. An instance of class `MaskedArray` can be thought of as containing the following parts:

- An array of data, of any shape;
- A mask of ones and zeros of the same shape as the data; and,
- A “fill value” — this is a value that may be used to replace the invalid entries in order to return a plain `numarray` array. The chief method that does this is the method `filled` discussed below.

We will use the terms “invalid value” and “invalid entry” to refer to the data value at a place corresponding to a mask value of 1. It should be emphasized that the invalid values are *never* used in any computation, and that the fill value is not used for *any* computational purpose. When an instance `x` of class `MaskedArray` is converted to its string representation, it is the result returned by `filled(x)` that is converted to a string.

### 14.3.1 Attributes of masked arrays

#### **flat**

(deprecated) Returns the masked array as a one-dimensional one. This is provided for compatibility with `numarray.ravel` is preferred. `flat` can be assigned to: ‘`x.flat = value`’ will change the values of `x`.

#### **real**

Returns the real part of the array if complex. It can be assigned to: ‘`x.real = value`’ will change the real parts of `x`.

#### **imaginary**

Returns the imaginary part of the array if complex. It can be assigned to: ‘`x.imaginary = value`’ will change the imaginary parts of `x`.

**shape**

The shape of a masked array can be accessed or changed by using the special attribute `shape`, as with `numarray` arrays. It can be assigned to: `'x.shape = newshape'` will change the shape of `x`. The new shape has to describe the same total number of elements.

**shared\_data**

This read-only flag if true indicates that the masked array shared a reference with the original data used to construct it at the time of construction. Changes to the original array will affect the masked array. (This is not the default behavior; see “Copying or not”.) This flag is informational only.

**shared\_mask**

This read-only flag if true indicates that the masked array *currently* shares a reference to the mask used to create it. Unlike `shared_data`, this flag may change as the result of modifying the array contents, as the mask uses copy on write semantics if it is shared.

## 14.3.2 Methods on masked arrays

**\_\_array\_\_**(*A*)

special method allows conversion to a `numarray` array if no element is actually masked. If there is a masked element, an `MAError` exception is thrown. Many `numarray` functions, such as `numarray.sqrt`, will attempt this conversion on their arguments. See also module function `filled` in section ??.

```
yn = numarray.array(x)
```

**astype**(*type*)

Return *self* as array of given *type*.

```
y = x.astype(Float32)
```

**byte\_swapped**( )

Returns the raw data `numarray` byte-swapped; included for consistency with `numarray` but probably meaningless.

```
y = x.byte_swapped()
```

**compressed**( )

Return an array of the valid elements. Result is one-dimensional.

```
y = x.compressed()
```

**count**(*axis=None*)

If *axis* is `None` return the count of non-masked elements in the whole array. Otherwise return an array of such counts along the axis given.

```
n = x.count()
y = x.count(0)
```

**fill\_value**( )

Get the current fill value.

```
v = x.fill_value()
```

**filled**(*fill\_value=None*)

Returns a `numarray` array with the masked values replaced by the fill value. See also the description of module function `filled` in section ??.

```
yn = x.filled()
```

**ids**()

Return the ids of the data and mask areas.

```
id1, id2 = x.ids()
```

**iscontiguous**()

Is the data area contiguous? See `numarray.scontiguous` in section 8.

```
if x.iscontiguous():
```

**itemsize**()

Size of individual data items in bytes. ‘`n = x.itemsize()`’

**mask**()

Return the data mask, or `None`.

```
m = x.mask()
```

**put**(*values*)

Set the value at each non-masked entry to the corresponding entry in *values*. The mask is unchanged. See also module function `put`.

```
x.put(values)
```

**putmask**(*values*)

Eliminate any masked values by setting the value at each masked entry to the corresponding entry in *values*. Set the mask to `None`.

```
x.putmask(values)
assert getmask(x) is None
```

**raw\_data**()

A reference to the non-filled data; portions may be meaningless. Expert use only.

```
d = x.raw_data ()
```

**savespace**(*v*)

Set the `spacesaver` attribute to *v*.

```
x.savespace (1)
```

**set\_fill\_value**(*v*)

Set the fill value to *v*. Omit *v* to restore default. ‘`x.set_fill_value(1.e21)`’

**set\_shape**(*args...*)

Set the shape.

```
x.set_shape (3, 12)
```

**size**(*axis*)

Number of elements in array, or along a particular *axis*.

```
totalsize = x.size ()
col_len = x.size (1)
```

**spacesaver**()

Query the spacesave flag.

```
flag = x.spacesaver()
```

**tolist**(*fill\_value=None*)

Return the Python list `self.filled(fill_value).tolist()`; note that masked values are filled.

```
alist=x.tolist()
```

**tostring**(*fill\_value=None*)

Return the string `self.filled(fill_value).tostring()` `s = x.tostring()`

**typecode**()

Return the type of the data. See module Precision, section ??.

```
z = x.typecode()
```

**unmask**()

Replaces the mask by None if possible. Subsequent operations may be faster if the array previously had an all-zero mask.

```
x.unmask()
```

**unshare\_mask**()

If `shared_mask` is currently true, replaces the reference to it with a copy.

```
x.unshare_mask()
```

### 14.3.3 Constructing masked arrays

**array**(*data, type=None, copy=1, savespace=0, mask=None, fill\_value=None*)

Creates a masked array with the given *data* and *mask*. The name *array* is simply an alias for the class name, *MaskedArray*. The fill value is set to *fill\_value*, and the *savespace* flag is applied. If *data* is a *MaskedArray*, its mask, typecode, spacesaver flag, and *fill\_value* will be used unless specifically overridden by one of the remaining arguments. In particular, if *d* is a masked array, `array(d, copy=0)` is *d*.

**masked\_array**(*data, mask=None, fill\_value=None*)

This is an easier-to-use version of *array*, for the common case of `typecode = None, copy = 0`. When *data* is newly-created this function can be used to make it a masked array without copying the data if *data* is already a *numarray* array.

**masked\_values**(*data, value, rtol=1.e-5, atol=1.e-8, type=None, copy=1, savespace=0*)

Constructs a masked array whose mask is set at those places where

$$\text{abs}(data - value) < atol + rtol * \text{abs}(data) \quad (14.1)$$

That is a careful way of saying that those elements of the *data* that have a value of *value* (to within a tolerance) are to be treated as invalid. If *data* is not of a floating point type, calls *masked\_object* instead.

**masked\_object**(*data*, *value*, *copy*=1, *savespace*=0)

Creates a masked array with those entries marked invalid that are equal to *value*. Again, *copy* and */savespace* are passed on to the `numarray` array constructor.

**asarray**(*data*, *type*=None)

This is the same as `array(data, typecode, copy=0)`. It is a short way of ensuring that something is an instance of `MaskedArray` of a given *type* before proceeding, as in `'data = asarray(data)'`.

If *data* already is a masked array and *type* is None then the return value is *data*; nothing is copied in that case.

**masked\_where**(*condition*, *data*, *copy*=1)

Creates a masked array whose shape is that of *condition*, whose values are those of *data*, and which is masked where elements of *condition* are true.

**masked**

This is a module constant that represents a scalar masked value. For example, if *x* is a masked array and a particular location such as `x[1]` is masked, the quantity `x[1]` will be this special constant. This special element is discussed more fully in section ?? “The constant masked”.

The following additional constructors are provided for convenience.

**masked\_equal**(*data*, *value*, *copy*=1)

**masked\_greater**(*data*, *value*, *copy*=1)

**masked\_greater\_equal**(*data*, *value*, *copy*=1)

**masked\_less**(*data*, *value*, *copy*=1)

**masked\_less\_equal**(*data*, *value*, *copy*=1)

**masked\_not\_equal**(*data*, *value*, *copy*=1)

`masked_greater` is equivalent to `masked_where(greater(data, value), data)`. Similarly, `masked_greater_equal`, `masked_equal`, `masked_not_equal`, `masked_less`, `masked_less_equal` are called in the same way with the obvious meanings. Note that for floating point data, `masked_values` is preferable to `masked_equal` in most cases.

**masked\_inside**(*data*, *v1*, *v2*, *copy*=1)

Creates an array with values in the closed interval [*v1*, *v2*] masked. *v1* and *v2* may be in either order.

**masked\_outside**(*data*, *v1*, *v2*, *copy*=1)

Creates an array with values outside the closed interval [*v1*, *v2*] masked. *v1* and *v2* may be in either order.

On entry to any of these constructors, *data* must be any object which the `numarray` package can accept to create an array (with the desired *type*, if specified). The *mask*, if given, must be None or any object that can be turned into a `numarray` array of integer type (it will be converted to type `MaskType`, if necessary), have the same shape as *data*, and contain only values of 0 or 1.

If the *mask* is not None but its shape does not match that of *data*, an exception will be thrown, unless one of the two is of length 1, in which case the scalar will be resized (using `numarray.resize`) to match the other.

See section 14.3.7 “Copying or not” for a discussion of whether or not the resulting array shares its data or its mask with the arguments given to these constructors.

**Important Tip** `filled` is very important. It converts its argument to a plain `numarray` array.

**filled**(*x*, *value*=None)

Returns *x* with any invalid locations replaced by a fill *value*. `filled` is guaranteed to return a plain `numarray` array. The argument *x* does not have to be a masked array or even an array, just something that `numarray/MA` can turn into one.

•If *x* is not a masked array, and not a `numarray` array, `numarray.array(x)` is returned.

- If *x* is a contiguous `numarray` array then *x* is returned. (A `numarray` array is contiguous if its data storage region is laid out in column-major order; `numarray` allows non-contiguous arrays to exist but they are not allowed in certain operations).
- If *x* is a masked array, but the mask is `None`, and *x*'s data array is contiguous, then it is returned. If the data array is not contiguous, a (contiguous) copy of it is returned.
- If *x* is a masked array with an actual mask, then an array formed by replacing the invalid entries with *value*, or `fill_value(x)` if *value* is `None`, is returned. If the fill value used is of a different type or precision than *x*, the result may be of a different type or precision than *x*.

Note that a new array is created only if necessary to create a correctly filled, contiguous, `numarray` array.

The function `filled` plays a central role in our design. It is the “exit” back to `numarray`, and is used whenever the invalid values must be replaced before an operation. For example, adding two masked arrays *a* and *b* is roughly:

```
masked_array(filled(a, 0) + filled(b, 0), mask_or(getmask(a), getmask(b)))
```

That is, fill the invalid entries of *a* and *b* with zeros, add them up, and declare any entry of the result invalid if either *a* or *b* was invalid at that spot. The functions `getmask` and `mask_or` are discussed later.

`filled` also can be used to simply be certain that some expression is a contiguous `numarray` array at little cost. If its argument is a `numarray` array already, it is returned without copying.

If you are certain that a masked array *x* contains a mask that is `None` or is all zeros, you can convert it to a Numeric array with the `numarray.array(x)` constructor. If you turn out to be wrong, an `MAError` exception is raised.

**`fill_value(x)`**

**`fill_value()`**

`fill_value(x)` and the method `x.fill_value()` on masked arrays, return a value suitable for filling *x* based on its type. If *x* is a masked array, then `x.fill_value()` results. The returned value for a given type can be changed by assigning to the following names in module `MA`. They should be set to scalars or one element arrays.

```
default_real_fill_value = numarray.array([1.0e20], Float32)
default_complex_fill_value = numarray.array([1.0e20 + 0.0j], Complex32)
default_character_fill_value = masked
default_integer_fill_value = numarray.array([0]).astype(UnsignedInt8)
default_object_fill_value = masked
```

The variable *masked* is a module variable of `MA` and is discussed in section 14.6.1. Calling `filled` with a *fill\_value* of masked sometimes produces a useful printed representation of a masked array. The function `fill_value` works on any kind of object.

`set_fill_value(a, fill_value)` is the same as `a.set_fill_value(fill_value)` if *a* is a masked array; otherwise it does nothing. Please note that the fill value is mostly cosmetic; it is used when it is needed to convert the masked array to a plain `numarray` array but not involved in most operations. In particular, setting the *fill\_value* to `1.e20` will *not*, repeat not, cause elements of the array whose values are currently `1.e20` to be masked. For that sort of behavior use the `masked_value` constructor.

#### 14.3.4 What are masks?

Masks are either `None` or 1-byte `numarray` arrays of 1's and 0's. To avoid excessive performance penalties, mask arrays are never checked to be sure that the values are 1's and 0's, and supplying a *mask* argument to a constructor with an illegal mask will have undefined consequences later.

*Masks have the `savespace` attribute set.* This attribute, discussed in part I, may have surprising consequences if you attempt to do any operations on them other than those supplied by this package. In particular, do not add or multiply

a quantity involving a mask. For example, if *m* is a mask consisting of 1080 1 values, `sum(m)` is 56, not 1080. Oops.

### 14.3.5 Working with masks

**is\_mask**(*m*)

Returns true if *m* is of a type and precision that would be allowed as the mask field of a masked array (that is, it is an array of integers with `numarray`'s `MaskType`, or it is `None`). To be a legal mask, *m* should contain only zeros or ones, but this is not checked.

**make\_mask**(*m*, *copy*=0, *flag*=0)

Returns an object whose entries are equal to *m* and for which `is_mask` would return true. If *m* is already a mask or `None`, it returns *m* or a copy of it. Otherwise it will attempt to make a mask, so it will accept any sequence of integers for *m*. If *flag* is true, `make_mask` returns `None` if its return value otherwise would contain no true elements. To make a legal mask, *m* should contain only zeros or ones, but this is not checked.

**make\_mask\_none**(*s*)

Returns a mask of all zeros of shape *s* (deprecated name: `create_mask`).

**getmask**(*x*)

Returns `x.mask()`, the mask of *x*, if *x* is a masked array, and `None` otherwise. **Note:** `getmask` may return `None` if *x* is a masked array but has a mask of `None`. (Please see caution above about operating on the result).

**getmaskarray**(*x*)

Returns `x.mask()` if *x* is a masked array and has a mask that is not `None`; otherwise it returns a zero mask array of the same shape as *x*. Unlike `getmask`, `getmaskarray` always returns an `numarray` array of typecode `MaskType`. (Please see caution above about operating on the result).

**mask\_or**(*m1*, *m2*)

Returns an object which when used as a mask behaves like the element-wise “logical or” of *m1* and *m2*, where *m1* and *m2* are either masks or `None` (e.g., they are the results of calling `getmask`). A `None` is treated as everywhere false. If both *m1* and *m2* are `None`, it returns `None`. If just one of them is `None`, it returns the other. If *m1* and *m2* refer to the same object, a reference to that object is returned.

### 14.3.6 Operations

Masked arrays support the operators `+`, `*`, `/`, `-`, `**`, and unary plus and minus. The other operand can be another masked array, a scalar, a `numarray` array, or something `numarray.array` can convert to a `numarray` array. The results are masked arrays.

In addition masked arrays support the in-place operators `+=`, `-=`, `*=`, and `/=`. Implementation of in-place operators differs from `numarray` semantics in being more generous about converting the right-hand side to the required type: any kind or lesser type accepted via an `astype` conversion. In-place operators truly operate in-place when the target is not masked.

### 14.3.7 Copying or not?

Depending on the arguments results of constructors may or may not contain a separate copy of the data or mask arguments. The easiest way to think about this is as follows: the given field, be it data or a mask, is required to be a `numarray` array, possibly with a given typecode, and a mask's shape must match that of the data. If the copy argument is zero, and the candidate array otherwise qualifies, a reference will be made instead of a copy. If for any reason the data is unsuitable as is, an attempt will be made to make a copy that is suitable. Should that fail, an exception will be thrown. Thus, a `copy=0` argument is more of a hope than a command.

If the basic array constructor is given a masked array as the first argument, its mask, typecode, spacesaver flag, and fill value will be used unless specifically specified by one of the remaining arguments. In particular, if *d* is a masked

`array, array(d, copy=0)` is *d*.

Since the default behavior for masks is to use a reference if possible, rather than a copy, which produces a sizeable time and space savings, it is especially important not to modify something you used as a mask argument to a masked array creation routine, if it was a `numarray` array of typecode `MaskType`.

### 14.3.8 Behaviors

`float(a)`

`int(a)`

`repr(a)`

`str(a)`

A masked array defines the conversion operators `str`, `repr`, `float`, and `int` by applying the corresponding operator to the `numarray` array `filled(a)`

### 14.3.9 Indexing and Slicing

Indexing and slicing differ from Numeric: while generally the same, they return a copy, not a reference, when used in an expression that produces a non-scalar result. Consider this example:

```
from Numeric import *
x = array([1.,2.,3.])
y = x[1:]
y[0] = 9.
print x
```

This will print `[1., 9., 3.]` since `x[1:]` returns a reference to a portion of *x*. Doing the same operation using MA,

```
from MA import *
x = array([1.,2.,3.])
y = x[1:]
y[0] = 9.
print x
```

will print `[1., 2., 3.]`, while *y* will be a separate array whose present value would be `[9., 3.]`. While sentiment on the correct semantics here is divided amongst the Numeric Python community as a whole, it is not divided amongst the author's community, on whose behalf this package is written.

### 14.3.10 Indexing in assignments

Using multiple sets of square brackets on the left side of an assignment statement will not produce the desired result:

```
x = array([[1,2],[3,4]])
x[1][1] = 20.                # Error, does not change x
x[1,1] = 20.                 # Correct, changes x
```

The reason is that `x[1]` is a copy, so changing it changes that copy, not *x*. Always use just one single square bracket for assignments.



### 14.3.11 Operations that produce a scalar result

If indexing or another operation on a masked array produces a scalar result, then a scalar value is returned rather than a one-element masked array. This raises the issue of what to return if that result is masked. The answer is that the module constant `masked` is returned. This constant is discussed in section 14.6.1. While this most frequently occurs from indexing, you can also get such a result from other functions. For example, averaging a 1-D array, all of whom's values are invalid, would result in `masked`.

### 14.3.12 Assignment to elements and slices

Assignment of a normal value to a single element or slice of a masked array has the effect of clearing the mask in those locations. In this way previously invalid elements become valid. The value being assigned is filled first, so that you are guaranteed that all the elements on the left-hand side are now valid.

Assignment of `None` to a single element or slice of a masked array has the effect of setting the mask in those locations, and the locations become invalid.

Since these operations change the mask, the result afterwards will no longer share a mask, since masks have copy-on-write semantics.

## 14.4 MaskedArray Attributes

**e**

**pi**

**NewAxis**

Constants `e`, `pi`, `NewAxis` from `numarray`, and the constants from module `Precision` that define nice names for the typecodes.

The special variables `masked` and `masked_print_option` are discussed in section 14.6.1.

The module `numarray` is an element of `MA`, so after `'from MA import *'`, you can refer to the functions in `numarray` such as `numarray.ones`; see part I for the constants available in `numarray`.

## 14.5 MaskedArray Functions

Each of the operations discussed below returns an instance of `MA` class `MaskedArray`, having performed the desired operation element-wise. In most cases the array arguments can be masked arrays or `numarray` arrays or something that `numarray` can turn into a `numarray` array, such as a list of real numbers.

In most cases, if `numarray` has a function of the same name, the behavior of the one in `MA` is the same, except that it “respects” the mask.

### 14.5.1 Unary functions

The result of a unary operation will be masked wherever the original operand was masked. It may also be masked if the argument is not in the domain of the function. The following functions have their standard meaning:

`absolute`, `arccos`, `arcsin`, `arctan`, `around`, `conjugate`, `cos`, `cosh`, `exp`, `floor`, `log`,  
`log10`, `negative` (also as operator `-`), `nonzero`, `sin`, `sinh`, `sqrt`, `tan`, `tanh`.

**fabs**(*x*)

The absolute value of *x* as a Float32 array.

## 14.5.2 Binary functions

Binary functions return a result that is masked wherever either of the operands were masked; it may also be masked where the arguments are not in the domain of the function.

add (also as operator +), subtract (also as operator -), multiply (also as operator \*), divide (also as operator /), power (also as operator \*\*), remainder, fmod, hypot, arctan2, bitwise\_and, bitwise\_or, bitwise\_xor.

## 14.5.3 Comparison operators

To compare arrays, use the following binary functions. Each of them returns a masked array of 1's and 0's.

equal, greater, greater\_equal, less, less\_equal, not\_equal.

Note that as in `numarray`, you can use a scalar for one argument and an array for the other. **Note:** See section ?? why operators and comparison functions are not exactly equivalent.

## 14.5.4 Logical operators

Arrays of logical values can be manipulated with:

logical\_and, logical\_not (unary), logical\_or, logical\_xor.

**alltrue**(*x*)

Returns 1 if all elements of *x* are true. Masked elements are treated as true.

**sometrue**(*x*)

Returns 1 if any element of *x* is true. Masked elements are treated as false.

## 14.5.5 Special array operators

**isarray**(*x*)

Return true *x* is a masked array.

**rank**(*x*)

The number of dimensions in *x*.

**shape**(*x*)

Returns the shape of *x*, a tuple of array extents.

**resize**(*x*, *shape*)

Returns a new array with specified *shape*.

**reshape**(*x*, *shape*)

Returns a copy of *x* with the given new *shape*.

**ravel**(*x*)

Returns *x* as one-dimensional MaskedArray.

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

Concatenates the arrays *a0*, ... *an* along the specified *axis*.

**repeat** (*a*, *repeats*, *axis=0*)

Repeat elements *i* of *a* *repeats*[*i*] times along *axis*. *repeats* is a sequence of length *a*.shape[*axis*] telling how many times to repeat each element.

**identity** (*n*)

Returns the identity matrix of shape *n* by *n*.

**indices** (*dimensions*, *type=None*)

Returns an array representing a grid of indices with row-only and column-only variation.

**len** (*x*)

This is defined to be the length of the first dimension of *x*. This definition, peculiar from the array point of view, is required by the way Python implements slicing. Use *size* for the total length of *x*.

**size** (*x*, *axis=None*)

This is the total size of *x*, or the length of a particular dimension *axis* whose index is given. When *axis* is given the dimension of the result is one less than the dimension of *x*.

**count** (*x*, *axis=None*)

Count the number of (non-masked) elements in the array, or in the array along a certain *axis*. When *axis* is given the dimension of the result is one less than the dimension of *x*.

**arange** ( )

**arrayrange** ( )

**diagonal** ( )

**fromfunction** ( )

**ones** ( )

**zeros** ( )

are the same as in Numeric, but return masked arrays.

**sum** ( )

**product** ( )

are called the same way as count; the difference is that the result is the sum or product of the unmasked element.

**average** (*x*, *axis=0*, *weights=None*, *returned=0*)

Compute the average value of the non-masked elements of *x* along the selected *axis*. If *weights* is given, it must match the size and shape of *x*, and the value returned is:

$$\text{average} = \frac{\sum \text{weights}_i \cdot x_i}{\sum \text{weights}_i} \quad (14.2)$$

In computing these sums, elements that correspond to those that are masked in *x* or *weights* are ignored. If successful a 2-tuple consisting of the average and the sum of the weights is returned.

**allclose** (*x*, *y*, *fill\_value=1*, *rtol=1.e-5*, *atol=1.e-8*)

Test whether or not arrays *x* and *y* are equal subject to the given relative and absolute tolerances. If *fill\_value* is 1, masked values are considered equal, otherwise they are considered different. The formula used for elements where both *x* and *y* have a valid value is:

$$|x - y| < \text{atol} + \text{rtol} \cdot |y| \quad (14.3)$$

This means essentially that both elements are small compared to *atol* or their difference divided by their value is small compared to *rtol*.

**allequal** (*x*, *y*, *fill\_value=1*)

This function is similar to `allclose`, except that exact equality is demanded. **Note:** Consider the problems of floating-point representations when using this function on non-integer numbers arrays.

**take**(*a*, *indices*, *axis*=0)

Returns a selection of items from *a*. See the documentation of `numarray.take` in section 7.

**transpose**(*a*, *axes*=None)

Performs a reordering of the axes depending on the tuple of indices *axes*; the default is to reverse the order of the axes.

**put**(*a*, *indices*, *values*)

The opposite of `take`. The values of the array *a* at the locations specified in *indices* are set to the corresponding value of *values*. The array *a* must be a contiguous array. The argument *indices* can be any integer sequence object with values suitable for indexing into the flat form of *a*. The argument *values* must be any sequence of values that can be converted to the typecode of *a*.

```
>>> x = arange(6)
>>> put(x, [2,4], [20,40])
>>> print x
[ 0  1 20  3 40  5 ]
```

Note that the target array *a* is not required to be one-dimensional. Since it is contiguous and stored in row-major order, the array indices can be treated as indexing *as* elements in storage order.

The wrinkle on this for masked arrays is that if the locations being set by `put` are masked, the mask is cleared in those locations.

**choose**(*condition*, *t*)

This function has a result shaped like *condition*. *t* must be a tuple. Each element of the tuple can be an array, a scalar, or the constant element masked (See section 14.6.1). Each element of the result is the corresponding element of `t[i]` where *condition* has the value *i*. The result is masked where *condition* is masked or where the selected element is masked or the selected element of *t* is the constant masked.

**where**(*condition*, *x*, *y*)

Returns an array that is `filled(x)` where *condition* is true, `filled(y)` where the condition is false. One of *x* or *y* can be the constant element masked (See section 14.6.1). The result is masked where *condition* is masked, where the element selected from *x* or *y* is masked, or where *x* or *y* itself is the constant masked and it is selected.

**innerproduct**(*a*, *b*)

**dot**(*a*, *b*)

These functions work as in `numarray`, but missing values don't contribute. The result is always a masked array, possibly of length one, because of the possibility that one or more entries in it may be invalid since all the data contributing to that entry was invalid.

**outerproduct**(*a*, *b*)

Produces a masked array such that `result[i, j] = a[i] * b[j]`. The result will be masked where `a[i]` or `b[j]` is masked.

**compress**(*condition*, *x*, *dimension*=-1)

Compresses out only those valid values where *condition* is true. Masked values in *condition* are considered false.

**maximum**(*x*, *y*=None)

**minimum**(*x*, *y*=None)

Compute the maximum (minimum) valid values of *x* if *y* is None; with two arguments, they return the element-wise larger or smaller of valid values, and mask the result where either *x* or *y* is masked. If both arguments are scalars a scalar is returned.

**sort**(*x*, *axis*=-1, *value*=None)

Returns the array *x* sorted along the given axis, with masked values treated as if they have a sort value of *value* but locations containing *value* are masked in the result if *x* had a mask to start with. **Note:** Thus if *x* contains *value* at a non-masked spot, but has other spots masked, the result may not be what you want.

**argsort**(*x*, *axis*=-1, *fill\_value*=None)

This function is unusual in that it returns a `numarray` array, equal to `numarray.argsort(filled(x, fill_value), axis)`; this is an array of indices for sorting along a given axis.

## 14.5.6 Controlling the size of the string representations

**get\_print\_limit**( )

**set\_print\_limit**(*n*=0)

These functions are used to limit printing of large arrays; query and set the limit for converting arrays using `str` or `repr`.

If an array is printed that is larger than this, the values are not printed; rather you are informed of the type and size of the array. If *n* is zero, the standard `numarray` conversion functions are used.

When imported, MA sets this limit to 300, and the limit is also made to apply to standard `numarray` arrays as well.

## 14.6 Helper classes

This section discusses other classes defined in module MA.

**class MAError**(*T*)

this class inherits from `Exception`, used to raise exceptions in the MA module. Other exceptions are possible, such as errors from the underlying `numarray` module.

### 14.6.1 The constant masked

A constant named `masked` in MA serves several purposes.

1. When a indexing operation on an `MaskedArray` instance returns a scalar result, but the location indexed was masked, then `masked` is returned. For example, given a one-dimensional array *x* such that `x.mask()[3]` is 1, then `x[3]` is masked.
2. When `masked` is assigned to elements of an array via indexing or slicing, those elements become masked. So after `x[3] = masked`, `x[3]` is masked.
3. Some other operations that may return scalar values, such as `average`, may return `masked` if given only invalid data.
4. To test whether or not a variable is this element, use the `is` or `is not` operator, not `==` or `!=`.
5. Operations involving the constant `masked` may result in an exception. In operations, `masked` behaves as an integer array of shape ( ) with one masked element. For example, using `+` for illustration,
  - `masked + masked` is masked.
  - `masked + numeric scalar` or `numeric scalar + masked` is masked.
  - `masked + array` or `array + masked` is a masked array with all elements masked if array is of a numeric type. The same is true if array is a `numarray` array.

## 14.6.2 The constant `masked_print_option`

Another constant, `masked_print_option` controls what happens when masked arrays and the constant `masked` are printed:

**`display()`**

Returns a string that may be used to indicate those elements of an array that are masked when the array is converted to a string, as happens with the print statement.

**`set_display(string)`**

This functions can be used to set the string that is used to indicate those elements of an array that are masked when the array is converted to a string, as happens with the print statement.

**`enable(flag)`**

can be used to enable (*flag* = 1, default) the use of the display string. If disabled (*flag* = 0), the conversion to string becomes equivalent to `str(self.filled())`.

**`enabled()`**

Returns the state of the display-enabling flag.

### Example of masked behavior

```
>>> from MA import *
>>> x=arange(5)
>>> x[3] = masked
>>> print x
[0 ,1 ,2 ,-- ,4 ,]
>>> print repr(x)
array(data =
      [0,1,2,0,4,],
      mask =
      [0,0,0,1,0,],
      fill_value=0,)]
>>> print x[3]
--
>>> print x[3] + 1.0
--
>>> print masked + x
[-- ,-- ,-- ,-- ,-- ,]
>>> masked_print_option.enable(0)
>>> print x
[0,1,2,0,4,]
>>> print x + masked
[0,0,0,0,0,]
>>> print filled(x+masked, -99)
[-99,-99,-99,-99,-99,]
```

**`class masked_unary_function(f, fill=0, domain=None)`**

Given a unary array function *f*, give a function which when applied to an argument *x* returns *f* applied to the array `filled(x, fill)`, with a mask equal to `mask_or(getmask(x), domain(x))`.

The argument *domain* therefore should be a callable object that returns true where *x* is not in the domain of *f*.

The following domains are also supplied as members of module `MA`:

**`class domain_check_interval(a, b)(x)`**

Returns true where *x* < *a* or *y* > *b*.

**`class domain_tan(eps)`**

*x* This is true where `abs(cos(x)) < eps`, that is, a domain suitable for the tangent function.

```
class domain_greater(v)(x)
    True where  $x \leq v$ .
```

```
class domain_greater_equal(v)(x)
    True where  $x \leq v$ .
```

```
class masked_binary_function(f, fillx=0, filly=0)
    Given a binary array function f, masked_binary_function(f, fillx=0, filly=0) defines a function whose value at  $x$  is f(filled(x, fillx), filled(y, filly)) with a resulting mask of mask_or(getmask(x), getmask(y)). The values fillx and filly must be chosen so that (fillx, filly) is in the domain of f.
```

In addition, an instance of `masked_binary_function` has two methods defined upon it:

```
reduce(target, axis = 0)
```

```
accumulate(target, axis = 0)
```

```
outer(a, b)
```

These methods perform reduction, accumulation, and applying the function in an outer-product-like manner, as discussed in the section 5.1.2.

```
class domained_binary_function( )
```

This class exists to implement division-related operations. It is the same as `masked_binary_function`, except that a new second argument is a domain which is used to mask operations that would otherwise cause failure, such as dividing by zero. The functions that are created from this class are `divide`, `remainder(mod)`, and `fmod`.

The following domains are available for use as the domain argument:

```
class domain_safe_divide( )(x, y)
```

True where `absolute(x)*divide_tolerance > absolute(y)`. As the comments in the code say, *better ideas welcome*. The constant `divide_tolerance` is set to `1.e-35` in the source and can be changed by editing its value in 'MA.py' and reinstalling. This domain is used for the divide operator.

## 14.7 Examples of Using MA

### 14.7.1 Data with a given value representing missing data

Suppose we have read a one-dimensional list of elements named  $x$ . We also know that if any of the values are `1.e20`, they represent missing data. We want to compute the average value of the data and the vector of deviations from average.

```
>>> from MA import *
>>> x = array([0.,1.,2.,3.,4.])
>>> x[2] = 1.e20
>>> y = masked_values(x, 1.e20)
>>> print average(y)
2.0
>>> print y-average(y)
[ -2.00000000e+00, -1.00000000e+00,  --,  1.00000000e+00,
  2.00000000e+00,]
```

### 14.7.2 Filling in the missing data

Suppose now that we wish to print that same data, but with the missing values replaced by the average value.

```
>>> print filled(y, average(y))
```

### 14.7.3 Numerical operations

We can do numerical operations without worrying about missing values, dividing by zero, square roots of negative numbers, etc.

```
>>> from MA import *
>>> x=array([1., -1., 3., 4., 5., 6.], mask=[0,0,0,0,1,0])
>>> y=array([1., 2., 0., 4., 5., 6.], mask=[0,0,0,0,0,1])
>>> print sqrt(x/y)
[ 1.00000000e+00, --, --, 1.00000000e+00, --, --,]
```

Note that four values in the result are invalid: one from a negative square root, one from a divide by zero, and two more where the two arrays *x* and *y* had invalid data. Since the result was of a real type, the print command printed `str(filled(sqrt(x/y)))`.

### 14.7.4 Seeing the mask

There are various ways to see the mask. One is to print it directly, the other is to convert to the `repr` representation, and a third is get the mask itself. Use of `getmask` is more robust than `x.mask()`, since it will work (returning `None`) if *x* is a `numarray` array or list.

```
>>> x = arange(10)
>>> x[3:5] = masked
>>> print x
[0 ,1 ,2 ,-- ,-- ,5 ,6 ,7 ,8 ,9 ,]
>>> print repr(x)
*** Masked array, mask present ***
Data:
[0 ,1 ,2 ,-- ,-- ,5 ,6 ,7 ,8 ,9 ,]
Mask (fill value [0,])
[0,0,0,1,1,0,0,0,0,0,]
>>> print getmask(x)
[0,0,0,1,1,0,0,0,0,0,]
```

### 14.7.5 Filling it your way

If we want to print the data with `-1`'s where the elements are masked, we use `filled`.

```
>>> print filled(z, -1)
[ 1.,-1.,-1., 1.,-1.,-1.,]
```

### 14.7.6 Ignoring extreme values

Suppose we have an array *d* and we wish to compute the average of the values in *d* but ignore any data outside the range -100. to 100.



```
v = masked_outside(d, -100., 100.)
print average(v)
```

### 14.7.7 Averaging an entire multidimensional array

The problem with averaging over an entire array is that the average function only reduces one dimension at a time. So to average the entire array, `ravel` it first.

```
>>> x
*** Masked array, no mask ***
Data:
[[ 0, 1, 2,]
 [ 3, 4, 5,]
 [ 6, 7, 8,]
 [ 9,10,11,]]
>>> average(x)
*** Masked array, no mask ***
Data:
[ 4.5, 5.5, 6.5,]
>>> average(ravel(x))
5.5
```

# Random Numbers

The `numpy.random_array` module (in conjunction with the `numpy.random_array.ranlib` submodule) provides a high-level interface to `ranlib`, which provides a good quality C implementation of a random-number generator.

## 15.1 General functions

**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**( )

This 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.

**random**(*shape*=[])

The `random` function takes a *shape*, and returns an array of `Float` numbers between 0.0 and 1.0. Neither 0.0 nor 1.0 is ever returned by this function. The array is filled from the generator following the canonical array organization.

If no argument is specified, the function returns a single floating point number, not an array.

**Note:** See discussion of the `flat` attribute in section 9.

**uniform**(*minimum*, *maximum*, *shape*=[])

The `uniform` function returns an array of the specified *shape* and containing `Float` random numbers strictly between *minimum* and *maximum*.

If no *shape* is specified, a single value is returned.

**randint**(*minimum*, *maximum*, *shape*=[])

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*,) with its elements randomly permuted.

## 15.2 Special random number distributions

### 15.2.1 Random floating point number distributions

**beta**(*a*, *b*, *shape*=[])

The `beta` function returns an array of the specified *shape* that contains `Float` numbers  $\beta$ -distributed with  $\alpha$ -parameter *a* and  $\beta$ -parameter *b*.

If no *shape* is specified, a single number is returned.

**chi\_square**(*df*, *shape*=[])

The `chi_square` function returns an array of the specified *shape* that contains `Float` numbers with the  $\chi^2$ -distribution with *df* degrees of freedom.

If no *shape* is specified, a single number is returned.

**exponential**(*mean*, *shape*=[])

The `exponential` function returns an array of the specified *shape* that contains `Float` numbers exponentially distributed with the specified *mean*.

If no *shape* is specified, a single number is returned.

**F**(*dfn*, *dfd*, *shape*=[])

The `F` function returns an array of the specified *shape* that contains `Float` numbers with the F-distribution with *dfn* degrees of freedom in the numerator and *dfd* degrees of freedom in the denominator.

If no *shape* is specified, a single number is returned.

**gamma**(*a*, *r*, *shape*=[])

The `gamma` function returns an array of the specified *shape* that contains `Float` numbers  $\beta$ -distributed with location parameter *a* and distribution shape parameter *r*.

If no *shape* is specified, a single number is returned.

**multivariate\_normal**(*mean*, *covariance*, *shape*=[])

The `multivariate_normal` function takes a one dimensional array argument *mean* and a two dimensional array argument *covariance*. Suppose the shape of *mean* is  $(n,)$ . Then the shape of *covariance* must be  $(n, n)$ . The function returns an array of `Floats`.

The effect of the *shape* parameter is:

- If no *shape* is specified, then an array with shape  $(n,)$  is returned containing a vector of numbers with a multivariate normal distribution with the specified mean and covariance.
- If *shape* is specified, then an array of such vectors is returned. The shape of the output is `shape.append((n,))`. The leading indices into the output array select a multivariate normal from the array. The final index selects one number from within the multivariate normal.

In either case, the behavior of `multivariate_normal` is undefined if *covariance* is not symmetric and positive definite.

**normal**(*mean*, *stddev*, *shape*=[])

The `normal` function returns an array of the specified *shape* that contains `Float` numbers normally distributed with the specified *mean* and standard deviation *stddev*.

If no *shape* is specified, a single number is returned.

**noncentral\_chi\_square**(*df*, *nonc*, *shape*=[])

The `noncentral_chi_square` function returns an array of the specified *shape* that contains `Float` numbers with the  $\chi^2$ -distribution with *df* degrees of freedom and noncentrality parameter *nonc*.

If no *shape* is specified, a single number is returned.

**noncentral\_F**(*dfn*, *dfd*, *nconc*, *shape*=[])

The `noncentral_F` function returns an array of the specified *shape* that contains `Float` numbers with the F-distribution with *dfn* degrees of freedom in the numerator, *dfd* degrees of freedom in the denominator, and noncentrality parameter *nconc*.

If no *shape* is specified, a single number is returned.

**standard\_normal**(*shape*=[])

The `standard_normal` function returns an array of the specified *shape* that contains `Float` numbers normally (Gaussian) distributed with mean zero and variance and standard deviation one.

If no *shape* is specified, a single number is returned.

## 15.2.2 Random integer number distributions

**binomial**(*trials*, *prob*, *shape*=[])

The `binomial` function returns an array with the specified *shape* that contains `Integer` numbers with the binomial distribution with *trials* and event probability *prob*. In other words, each value in the returned array is the number of times an event with probability *prob* occurred within *trials* repeated trials.

If no *shape* is specified, a single number is returned.

**negative\_binomial**(*trials*, *prob*, *shape*=[])

The `negative_binomial` function returns an array with the specified *shape* that contains `Integer` numbers with the negative binomial distribution with *trials* and event probability *prob*.

If no *shape* is specified, a single number is returned.

**multinomial**(*trials*, *probs*, *shape*=[])

The `multinomial` function returns an array with that contains integer numbers with the multinomial distribution with *trials* and event probabilities given in *probs*. *probs* must be a one dimensional array. There are `len(probs)+1` events. `probs[i]` is the probability of the *i*-th event for  $0 \leq i < \text{len}(\text{probs})$ . The probability of event `len(probs)` is `1.-Numeric.sum(probs)`.

The function returns an integer array of shape `shape + (len(probs)+1,)`. If *shape* is not specified this is one multinomially distributed vector of shape `(len(prob)+1,)`. Otherwise each `returnarray[i, j, ..., :]` is an integer array of shape `(len(prob)+1,)` containing one multinomially distributed vector.

**poisson**(*mean*, *shape*=[])

The `poisson` function returns an array with the specified *shape* that contains `Integer` numbers with the Poisson distribution with the specified *mean*.

If no *shape* is specified, a single number is returned.

## 15.3 Examples

Some example uses of the `numarray.random_array` module. **Note:** Naturally the exact output of running these examples will be different each time!

```

>>> from numpy.random_array 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

```

Most of the functions in this package take zero or more distribution specific parameters plus an optional *shape* parameter. The *shape* parameter gives the shape of the output array:

```

>>> from numpy.random_array import *
>>> print standard_normal()
-0.435568600893
>>> print standard_normal(5)
[-1.36134553  0.78617644 -0.45038718  0.18508556  0.05941355]
>>> print standard_normal((5,2))
[[ 1.33448863 -0.10125473]
 [ 0.66838062  0.24691346]
 [-0.95092064  0.94168913]
 [-0.23919107  1.89288616]
 [ 0.87651485  0.96400219]]
>>> print normal(7., 4., (5,2)) #mean=7, std. dev.=4
[[ 2.66997623 11.65832615]
 [ 6.73916003  6.58162862]
 [ 8.47180378  4.30354905]
 [ 1.35531998 -2.80886841]
 [ 7.07408469 11.39024973]]
>>> print exponential(10., 5) #mean=10
[ 18.03347754  7.11702306  9.8587961  32.49231603 28.55408891]
>>> print beta(3.1, 9.1, 5) # alpha=3.1, beta=9.1
[ 0.1175056  0.17504358  0.3517828  0.06965593  0.43898219]
>>> print chi_square(7, 5) # 7 degrees of freedom (dfs)
[ 11.99046516  3.00741053  4.72235727  6.17056274  8.50756836]
>>> print noncentral_chi_square(7, 3, 5) # 7 dfs, noncentrality 3
[ 18.28332138  4.07550335 16.0425396  9.51192093  9.80156231]
>>> F(5, 7, 5) # 5 and 7 dfs
array([ 0.24693671,  3.76726145,  0.66883826,  0.59169068,  1.90763224])
>>> noncentral_F(5, 7, 3., 5) # 5 and 7 dfs, noncentrality 3
array([ 1.17992553,  0.7500126 ,  0.77389943,  9.26798989,  1.35719634])
>>> binomial(32, .5, 5) # 32 trials, prob of an event = .5
array([12, 20, 21, 19, 17])
>>> negative_binomial(32, .5, 5) # 32 trials: prob of an event = .5
array([21, 38, 29, 32, 36])

```

Two functions that return generate multivariate random numbers (that is, random vectors with some known relationship between the elements of each vector, defined by the distribution). They are `multivariate_normal` and `multinomial`. For these two functions, the lengths of the leading axes of the output may be specified. The length of the last axis is determined by the length of some other parameter.

```

>>> multivariate_normal([1,2], [[1,2],[2,1]], [2,3])
array([[[ 0.14157988,  1.46232224],
 [-1.11820295, -0.82796288],
 [ 1.35251635, -0.2575901 ]],
 [[-0.61142141,  1.0230465 ],
 [-1.08280948, -0.55567217],
 [ 2.49873002,  3.28136372]]])
>>> x = multivariate_normal([10,100], [[1,2],[2,1]], 10000)
>>> x_mean = sum(x)/10000
>>> print x_mean
[ 9.98599893 100.00032416]
>>> x_minus_mean = x - x_mean
>>> cov = matrixmultiply(transpose(x_minus_mean), x_minus_mean) / 9999.
>>> cov
array([[ 2.01737122,  1.00474408],
 [ 1.00474408,  2.0009806 ]])

```

The a priori probabilities for a multinomial distribution must sum to one. The prior probability argument to

`multinomial` doesn't give the prior probability of the last event: it is computed to be one minus the sum of the others.

```
>>> multinomial(16, [.1, .4, .2]) # prior probabilities [.1, .4, .2, .3]
array([2, 7, 1, 6])
>>> multinomial(16, [.1, .4, .2], [2,3]) # output shape [2,3,4]
array([[[ 1, 9, 1, 5],
[ 0, 10, 3, 3],
[ 4, 9, 3, 0]],
[[ 1, 6, 1, 8],
[ 3, 4, 5, 4],
[ 1, 5, 2, 8]]])
```

# Appendix





---

# Glossary

This chapter provides a glossary of terms.<sup>1</sup>

**array** An array refers to the Python object type defined by the NumPy extensions to store and manipulate numbers efficiently.

**byteswapped**

**discontiguous**

**misaligned**

**misbehaved array** A `numarray` which is byteswapped, misaligned, or discontiguous.

**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).

**ufunc** 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.

**universal function** See `ufunc`.

---

<sup>1</sup>Please let us know of any additions to this list which you feel would be helpful.



# INDEX

## Symbols

`*` (in module MA), 109  
`**` (in module MA), 109  
`*=`, 31  
`+`, 109  
`+=`, 31  
`-` (in module MA), 108, 109  
`/` (in module MA), 109  
`/=`, 31  
`__array__()` (MaskedArray method), 101

## A

`absolute` (in module MA), 108  
`accumulate()`  
    method, 41  
    masked\_binary\_function method, 114  
`add` (in module MA), 109  
`allclose()`  
    in module MA, 110  
    in module numarray, 60  
`allequal()` (in module MA), 110  
`alltrue()`  
    in module MA, 109  
    in module numarray, 59  
`arange()`  
    in module MA, 110  
    in module numarray, 24  
`arccos` (in module MA), 108  
`arcsin` (in module MA), 108  
`arctan` (in module MA), 108  
`arctan2` (in module MA), 109  
`argmax()` (in module numarray), 55  
`argmin()` (in module numarray), 55  
`argsort()`  
    in module MA, 112  
    in module numarray, 54  
`around` (in module MA), 108  
`array()`  
    in module numarray, 18  
    MaskedArray method, 103  
`array_repr()` (in module numarray), 58

`array_str()` (in module numarray), 58  
`arrayrange()`  
    in module MA, 110  
    in module numarray, 24  
`asarray()`  
    in module numarray, 18  
    MaskedArray method, 104  
`astype()`  
    method, 29  
    MaskedArray method, 101  
`average()` (in module MA), 110

## B

`beta()` (in module numarray.random\_array), 118  
`binomial()` (in module numarray.random\_array), 119  
`bitwise_and` (in module MA), 109  
`bitwise_or` (in module MA), 109  
`bitwise_xor` (in module MA), 109  
`Bool`, 19  
`boxcar()` (in module numarray.convolve), 89  
`broadcasting`, 30, 31  
`byte_swapped()` (MaskedArray method), 101  
`byteswap()` (numarray method), 62

## C

`cfft_b()` (in module numarray.fft.fftpack), 92  
`cfft_f()` (in module numarray.fft.fftpack), 92  
`cfft_i()` (in module numarray.fft.fftpack), 92  
`chi_square()` (in module numarray.random\_array), 118  
`cholesky_decomposition()` (in module numarray.linear\_algebra), 96  
`choose()`  
    in module MA, 111  
    in module numarray, 52  
`clip()` (in module numarray), 56  
`Complex32`, 19  
`Complex64`, 19  
`compress()`  
    in module MA, 111

- in module `numarray`, 53
- `compressed()` (`MaskedArray` method), 101
- `concatenate`, 22
- `concatenate()`
  - in module `MA`, 110
  - in module `numarray`, 57
- `conjugate` (in module `MA`), 108
- `convolve()` (in module `numarray.convolve`), 89
- `convolve2d()` (in module `numarray.convolve`), 89
- `copy()` (`numarray` method), 61
- `correlate()` (in module `numarray.convolve`), 89
- `correlate2d()` (in module `numarray.convolve`), 89
- `cos` (in module `MA`), 108
- `cosh` (in module `MA`), 108
- `count()`
  - in module `MA`, 110
  - `MaskedArray` method, 101
- `cumproduct()` (in module `numarray`), 59
- `cumsum()` (in module `numarray`), 59

## D

- `determinant()` (in module `numarray.linear_algebra`), 96
- `diagonal()`
  - in module `MA`, 110
  - in module `numarray`, 53, 58
- `display()` (`masked_print_option` method), 113
- `divide` (in module `MA`), 109
- `domain_check_interval` (class in `MA`), 113
- `domain_greater` (class in `MA`), 114
- `domain_greater_equal` (class in `MA`), 114
- `domain_safe_divide` (class in `MA`), 114
- `domain_tan` (class in `MA`), 113
- `domained_binary_function` (class in `MA`), 114
- `dot()`
  - in module `MA`, 111
  - in module `numarray`, 55
- Dubois, Paul F., 99

## E

- `e` (data in `MA`), 108
- `eigenvalues()` (in module `numarray.linear_algebra`), 96
- `eigenvectors()` (in module `numarray.linear_algebra`), 96
- `enable()` (`masked_print_option` method), 113
- `enabled()` (`masked_print_option` method), 113
- `equal` (in module `MA`), 109
- `exp` (in module `MA`), 108
- `exponential()` (in module `numarray.random_array`), 118

## F

- `F()` (in module `numarray.random_array`), 118
- `fabs()` (in module `MA`), 109
- `fft()` (in module `numarray.fft`), 91
- `fft2d()` (in module `numarray.fft`), 92
- `fill_value()`
  - in module `MA`, 105
  - `MaskedArray` method, 101, 105
- `filled()`
  - in module `MA`, 104
  - `MaskedArray` method, 102
- `flat`
  - `MaskedArray` attribute, 100
  - `numarray` attribute, 65
- `float()` (in module `MA`), 107
- `Float32`, 19
- `Float64`, 19
- `floor` (in module `MA`), 108
- `fmod` (in module `MA`), 109
- `fromfile()` (in module `numarray`), 55
- `fromfunction()`
  - in module `MA`, 110
  - in module `numarray`, 25
- `fromstring()` (in module `numarray`), 55
- `FULL` (data in `numarray.convolve`), 90

## G

- `gamma()` (in module `numarray.random_array`), 118
- `generalized_inverse()` (in module `numarray.linear_algebra`), 96
- `get_print_limit()` (in module `MA`), 112
- `get_seed()` (in module `numarray.random_array`), 117
- `getflat()` (`numarray` method), 63
- `getimag()` (`numarray` method), 63
- `getimaginary()` (`numarray` method), 63
- `getmask()` (in module `MA`), 106
- `getmaskarray()` (in module `MA`), 106
- `getreal()` (`numarray` method), 63
- `getshape`, 20
- `getshape()` (`numarray` method), 63
- `greater` (in module `MA`), 109
- `greater_equal` (in module `MA`), 109

## H

- `Heigenvalues()` (in module `numarray.linear_algebra`), 96
- `Heigenvectors()` (in module `numarray.linear_algebra`), 96
- `hypot` (in module `MA`), 109

## I

- `identity()`

- in module MA, 110
  - in module numarray, 27, 59
- ids() (MaskedArray method), 102
- imag (numarray attribute), 65
- imaginary
  - MaskedArray attribute, 100
  - numarray attribute, 66
- indices()
  - in module MA, 110
  - in module numarray, 56
- innerproduct()
  - in module MA, 111
  - in module numarray, 57
- inputarray() (in module numarray), 18
- int() (in module MA), 107
- Int16, 19
- Int32, 19
- Int64, 19
- Int8, 19
- inverse() (in module numarray.linear\_algebra), 96
- inverse\_fft() (in module numarray.fft), 92
- inverse\_real\_fft() (in module numarray.fft), 92
- is\_mask() (in module MA), 106
- isarray() (in module MA), 109
- iscontiguous()
  - MaskedArray method, 102
  - numarray method, 61
- itemsize()
  - MaskedArray method, 102
  - numarray method, 61

## L

- len() (in module MA), 110
- less (in module MA), 109
- less\_equal (in module MA), 109
- linear\_least\_squares() (in module numarray.linear\_algebra), 97
- log (in module MA), 108
- log10 (in module MA), 108
- logical\_and (in module MA), 109
- logical\_not (in module MA), 109
- logical\_or (in module MA), 109
- logical\_xor (in module MA), 109

## M

### MA

- \*, 109
- \*\*, 109
- +, 109
- , 108, 109
- /, 109
- constructor, 103, 104, 106

- create\_mask (deprecated),  
→make\_mask\_none
- default\_character\_fill\_value, 105
- default\_complex\_fill\_value, 105
- default\_integer\_fill\_value, 105
- default\_object\_fill\_value, 105
- default\_real\_fill\_value, 105
- divide\_tolerance, 114
- Installation, 99
- invalid, 99, 108
- mask, 106
- masked, 108, 112, 113
- masked (constant), 112
- masked\_binary\_function, 114
- masked\_print\_option (constant), 113
- MaskedArray, 100, 108
- set\_fill\_value, 105
- valid, 99, 108
- MA (extension module), **99**
- MAError (class in MA), 112
- make\_mask() (in module MA), 106
- make\_mask\_none() (in module MA), 106
- mask() (MaskedArray method), 102
- mask\_or() (in module MA), 106
- masked (data in MA), 104
- masked\_array() (MaskedArray method), 103
- masked\_binary\_function (class in MA), 114
- masked\_equal() (MaskedArray method), 104
- masked\_greater() (MaskedArray method), 104
- masked\_greater\_equal() (MaskedArray method), 104
- masked\_inside() (MaskedArray method), 104
- masked\_less() (MaskedArray method), 104
- masked\_less\_equal() (MaskedArray method), 104
- masked\_not\_equal() (MaskedArray method), 104
- masked\_object() (MaskedArray method), 104
- masked\_outside() (MaskedArray method), 104
- masked\_unary\_function (class in MA), 113
- masked\_values() (MaskedArray method), 103
- masked\_where() (MaskedArray method), 104
- MaskedArray, →MA
- masks, description of, 105
- masks, savespace attribute, 105
- matrixmultiply() (in module numarray), 55
- max() (numarray method), 64
- maximum() (in module MA), 111
- mean() (numarray method), 63
- min() (numarray method), 64
- minimum() (in module MA), 111
- multidimensional arrays, 19
- multidimensional indexing, 32

multinomial() (in module numarray.random\_array), 119  
 multiply (in module MA), 109  
 multivariate\_normal() (in module numarray.random\_array), 118

## N

NA\_ByteOrder(), 84  
 NA\_ComplexArrayCheck(), 85  
 NA\_copy(), 85  
 NA\_copyArray(), 85  
 NA\_elements(), 85  
 NA\_GET1(), 75  
 NA\_get1\_Complex64(), 77  
 NA\_get1\_Float64(), 76  
 NA\_get1\_Int64(), 76  
 NA\_GET1a(), 75  
 NA\_GET1b(), 75  
 NA\_get1D\_Complex64(), 79  
 NA\_get1D\_Float64(), 79  
 NA\_get1D\_Int64(), 79  
 NA\_GET1f(), 75  
 NA\_GET2(), 76  
 NA\_get2\_Complex64(), 77  
 NA\_get2\_Float64(), 76  
 NA\_get2\_Int64(), 76  
 NA\_GET2a(), 76  
 NA\_GET2b(), 76  
 NA\_GET2f(), 76  
 NA\_get\_Complex64(), 76  
 NA\_get\_Float64(), 76  
 NA\_get\_Int64(), 76  
 NA\_get\_offset(), 79  
 NA\_GETP(), 75  
 NA\_GETPa(), 75  
 NA\_GETPb(), 75  
 NA\_GETPf(), 75  
 NA\_getPythonScalar(), 85  
 NA\_IeeeSpecial32(), 84  
 NA\_IeeeSpecial64(), 84  
 NA\_InputArray(), 71  
 NA\_intTupleFromMaybeLongs(), 84  
 NA\_IoArray(), 71  
 NA\_isIntegerSequence(), 84  
 NA\_maxType(), 85  
 NA\_maybeLongsFromIntTuple(), 84  
 NA\_NDArrayCheck(), 85  
 NA\_NewAll(), 84  
 NA\_NewAllStrides(), 84  
 NA\_NewArray(), 83  
 NA\_NumArrayCheck(), 85  
 NA\_OptionalOutputArray(), 72  
 NA\_OutputArray(), 71  
 NA\_ReturnOutput(), 72

NA\_SET1(), 76  
 NA\_set1\_Complex64(), 77  
 NA\_set1\_Float64(), 76  
 NA\_set1\_Int64(), 76  
 NA\_SET1a(), 75  
 NA\_SET1b(), 75  
 NA\_set1D\_Complex64(), 79  
 NA\_set1D\_Float64(), 79  
 NA\_set1D\_Int64(), 79  
 NA\_SET1f(), 76  
 NA\_SET2(), 76  
 NA\_set2\_Complex64(), 77  
 NA\_set2\_Float64(), 76  
 NA\_set2\_Int64(), 76  
 NA\_SET2a(), 76  
 NA\_SET2b(), 76  
 NA\_SET2f(), 76  
 NA\_set\_Complex64(), 77  
 NA\_set\_Float64(), 76  
 NA\_set\_Int64(), 76  
 NA\_setArrayFromSequence(), 85  
 NA\_setFromPythonScalar(), 85  
 NA\_SETP(), 75  
 NA\_SETPa(), 75  
 NA\_SETPb(), 75  
 NA\_SETPf(), 75  
 NA\_ShapeEqual(), 84  
 NA\_ShapeLessThan(), 84  
 NA\_typeNoToName(), 84  
 NA\_typeNoToTypeObject(), 84  
 NA\_typeObjectToTypeNo(), 84  
 NA\_updateDataPtr(), 84  
 NA\_vNewArray(), 83  
 negative (in module MA), 108  
 negative\_binomial() (in module numarray.random\_array), 119  
 NewAxis (data in MA), 108  
 noncentral\_chi\_square() (in module numarray.random\_array), 118  
 noncentral\_F() (in module numarray.random\_array), 118  
 nonzero (in module MA), 108  
 nonzero() (in module numarray), 52  
 normal() (in module numarray.random\_array), 118  
 not\_equal (in module MA), 109  
 numarray  
   constructor, 23  
 numarray (extension module), 3  
 numarray.convolve (extension module), 89  
 numarray.fft (extension module), 91  
 numarray.fft.fftpack (extension module), 92  
 numarray.linear\_algebra (extension module), 95

numarray.random\_array (extension module),  
117

## O

observations, dealing with missing, 99

ones()

in module MA, 110

in module numarray, 23

outer()

method, 41

masked\_binary\_function method, 114

outerproduct()

in module MA, 111

in module numarray, 58

## P

PASS (data in numarray.convolve), 90

permutation() (in module numarray.random\_array), 117

pi (data in MA), 108

poisson() (in module numarray.random\_array),  
119

power (in module MA), 109

printing arrays, 22

product()

in module MA, 110

in module numarray, 59

Properties, 99

put, 36

put()

in module MA, 111

in module numarray, 50

MaskedArray method, 102

putmask()

in module numarray, 51

MaskedArray method, 102

PyArray\_As1D(), 82

PyArray\_As2D(), 82

PyArray\_CanCastSafely(), 83

PyArray\_Cast(), 83

PyArray\_Check(), 83

PyArray\_ContiguousFromObject(), 82

PyArray\_Copy(), 83

PyArray\_CopyFromObject(), 82

PyArray\_DescrFromType(), 83

PyArray\_Free(), 82

PyArray\_FromDims(), 82

PyArray\_FromDimsAndData(), 82

PyArray\_FromObject(), 82

PyArray\_isArray(PyObject \*o()), 83

PyArray\_NBYTES(), 83

PyArray\_Return(), 82

PyArray\_Size(), 83

## R

randint() (in module numarray.random\_array),  
117

random() (in module numarray.random\_array), 117

rank, 17, 20

rank() (in module MA), 109

ravel()

in module MA, 109

in module numarray, 52

raw\_data() (MaskedArray method), 102

real

MaskedArray attribute, 100

numarray attribute, 65

real\_fft() (in module numarray.fft), 92

real\_fft2d() (in module numarray.fft), 92

reduce()

method, 41

masked\_binary\_function method, 114

reduceat() (method), 42

remainder (in module MA), 109

repeat()

in module MA, 110

in module numarray, 51, 58

repr() (in module MA), 107

reshape()

in module MA, 109

in module numarray, 20

resize()

in module MA, 109

in module numarray, 23, 58

rfftb() (in module numarray.fft.fftpack), 93

rfftf() (in module numarray.fft.fftpack), 93

rffti() (in module numarray.fft.fftpack), 92

## S

SAME (data in numarray.convolve), 90

savespace() (MaskedArray method), 102

searchsorted() (in module numarray), 54

seed() (in module numarray.random\_array), 117

set\_display() (masked\_print\_option method),  
113

set\_fill\_value() (MaskedArray method), 102

set\_print\_limit() (in module MA), 112

set\_shape() (MaskedArray method), 102

setimag() (numarray method), 63

setimaginary() (numarray method), 63

setreal() (numarray method), 63

setshape, 20

setshape() (numarray method), 63

shape, 20

shape() (in module MA), 109

shape

MaskedArray attribute, 101

numarray attribute, 65



shared\_data (MaskedArray attribute), 101  
 shared\_mask (MaskedArray attribute), 101  
 sin (in module MA), 108  
 singular\_value\_decomposition() (in module numarray.linear\_algebra), 97  
 sinh (in module MA), 108  
 size()  
     in module MA, 110  
     MaskedArray method, 103  
 solve\_linear\_equations() (in module numarray.linear\_algebra), 97  
 sometrue()  
     in module MA, 109  
     in module numarray, 59  
 sort()  
     in module MA, 112  
     in module numarray, 54  
 spacesaver() (MaskedArray method), 103  
 sqrt (in module MA), 108  
 standard\_normal() (in module numarray.random\_array), 119  
 str() (in module MA), 107  
 stride, 34  
     Negative, 34  
 subtract (in module MA), 109  
 sum()  
     in module MA, 110  
     in module numarray, 59  
     numarray method, 63  
 swapaxes() (in module numarray), 56

## T

take, 36  
 take()  
     in module MA, 111  
     in module numarray, 49  
 tan (in module MA), 108  
 tanh (in module MA), 108  
 tofile() (numarray method), 62  
 tolist()  
     MaskedArray method, 103  
     numarray method, 62  
 tostring()  
     MaskedArray method, 103  
     numarray method, 62  
 trace() (in module numarray), 53  
 transpose()  
     in module MA, 111  
     in module numarray, 51  
 type() (numarray method), 61  
 type argument, 19  
 typecode() (MaskedArray method), 103

## U

UInt16, 19  
 UInt32, 19  
 UInt64, 19  
 UInt8, 19  
 unary, 113  
 uniform() (in module numarray.random\_array), 117  
 unmask() (MaskedArray method), 103  
 unshare\_mask() (MaskedArray method), 103

## V

VALID (data in numarray.convolve), 90  
 void NA\_get\_offset(), 76

## W

where()  
     in module MA, 111  
     in module numarray, 52

## Z

zeros()  
     in module MA, 110  
     in module numarray, 23