National Centre for Atmospheric Science
NATURAL ENVIRONMENT RESEARCH COUNCIL

Centre for Environmental Data Archival
SCIENCE AND TECHNOLOGY FACILITIES COUNCIL
NATURAL ENVIRONMENT RESEARCH COUNCIL

# Handling arrays in Python (numpy)

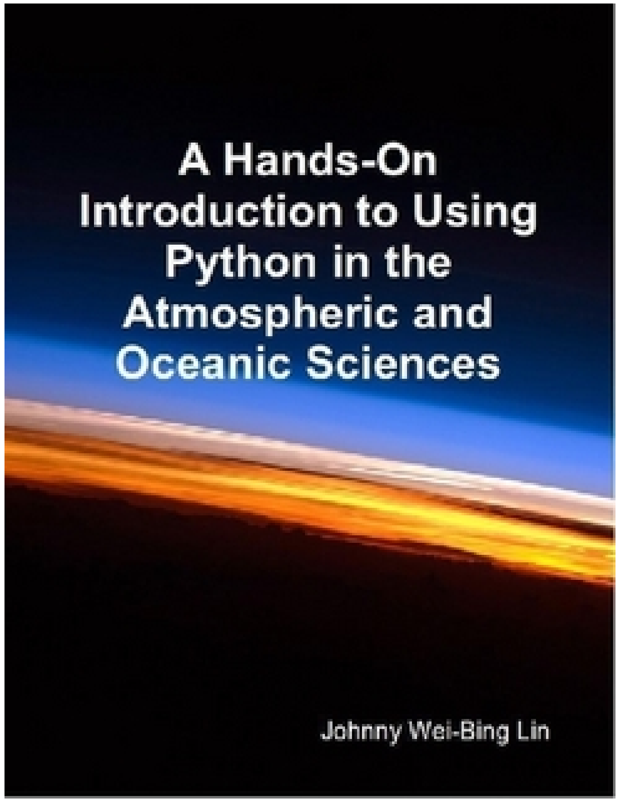
Thanks to all contributors:

Alison Pamment, Sam Pepler, Ag Stephens, Stephen Pascoe, Anabelle Guillory, Graham Parton, Esther Conway, Wendy Garland, Alan Iwi and Matt Pritchard.

# With special thanks to…

**Johnny Lin**, who writes a great python/atmospheric science blog, exercises, examples, presentations, books etc.

`Much of this is borrowed from...`


A Hands-On Introduction to Using Python in the Atmospheric and Oceanic Sciences
Johnny Wei-Bing Lin

http://pyaos.johnny-lin.com/?p=1256

## What is an array?

- An **array** is like a list except:
  - All elements are of the same type, so operations with arrays are much faster.
  - Multi-dimensional arrays are more clearly supported.
  - Array operations are supported

## The NumPy package

- NumPy is the standard array package in Python. (There are others, but the community has now converged on NumPy).
- NumPy is written in C so processing of large arrays is much faster than processing lists.
- To utilize NumPy's functions and attributes, you import the package `numpy`.
- Often NumPy is imported as an alias, e.g.:

```
import numpy as np
```

## Creating arrays

- Use the `array` function on a list:

```
import numpy as np
a = np.array([[2, 3, -5],[21, -2, 1]])
```

## Creating arrays

- Use the `array` function on a list:

```
import numpy as np
a = np.array([[2, 3, -5],[21, -2, 1]])
```

- The `array` function will match the array type to the contents of the list.

# Creating arrays

- To force a certain numerical type for the array, set the `dtype` keyword to a type code:

```
a = np.array([[2, 3, -5], [21, -2, 1]],
                              dtype = ‘d’)
```

National Centre for Atmospheric Science
NATURAL ENVIRONMENT RESEARCH COUNCIL

Centre for Environmental Data Archival
SCIENCE AND TECHNOLOGY FACILITIES COUNCIL
NATURAL ENVIRONMENT RESEARCH COUNCIL

## Typecodes for arrays

Some common typecodes (which are strings):

```
'd':  Double precision floating
'f':  Single precision floating
'i':  Short integer
'I':  Long integer
```

## Other ways of creating arrays

To create an array of a given shape filled with zeros, use the `zeros` function (with `dtype` being optional):

```
a = np.zeros((3,2), dtype='d')
```

To create an array the same as `range`, use the `arange` function (again `dtype` is optional):

```
a = np.arange(10)
```

## Array indexing

Like lists, element addresses start with zero, so the first element of 1-D array a is `a[0]`, the second is `a[1]`, etc.

Like lists, you can reference elements starting from the end, e.g., element `a[-1]` is the last element in a 1-D array.

## Array slicing

- Element addresses in a range are separated by a colon, e.g.: `a[4:8]`
- The lower limit is inclusive, and the upper limit is exclusive, e.g.: `a[1:4]` contains *second* to *fourth* values of a.

- If one of the limits is left out, the range is extended to the end of the range, e.g.: `a[:6]` contains the first 6 elements of a.
- To specify all elements use a colon only: `a[:]`

## Array indexing

For multi-dimensional arrays, indexing between different dimensions is separated by commas.

- The fastest varying dimension is the last index. Thus, a 2-D array is indexed [row, col].
- Slicing rules also work as applied for each dimension (e.g., a colon selects all elements in that dimension).

# Multi-dimensional array indexing

Consider the following example:

```
a = np.array([[2, 3.2, 5.5, -6.4, -2.2, 2.4],
      [1, 22, 4, 0.1, 5.3, -9],
      [3, 1, 2.1, 21, 1.1, -2]])
```

What is a[1,2] equal to? a[1,:]? a[1,1:4]?

```
a[1,2]   →  4
a[1,:]   →  [1, 22, 4, 0.1, 5.3, -9]
a[1,1:4] →  [22, 4, 0.1]
```

# Interrogating arrays

- Some information about arrays comes through functions (methods) on the array, others through attributes attached to the array.
- For this and the next slide, assume a and b are arrays and `NumPy` is imported as np.
- Shape of the array: `np.shape(a)`
- Rank of the array: `np.rank(a)`

## Interrogating arrays

- Number of elements in the array (do not use len for arrays):

```
np.size(a)
```

- Typecode of the array: `a.dtype.char`

- Get maximum or minimum value in the array:

```
arr.max()
arr.min()
```

## Array manipulation

| | |
|---|---|
| Reshape the array: e.g., | `np.reshape(a,(2,3))` |
| Transpose the array: | `np.transpose(a)` |
| Flatten to a 1-D array: | `np.ravel(a)` |
| Concatenate arrays: | `np.concatenate(a,b)` |
| Repeat array elements: e.g., | `np.repeat(a,3)` |

Convert array a to another type: `b = a.astype('f')`
where the argument is the `typecode` for b.

## meshgrid

A common task is to generate a pair of arrays which represent the coordinates of our data. When the orthogonal 1d coordinate arrays already exist, numpy's `meshgrid` function is very useful:

```
>>> x_g = np.linspace(0, 9, 3)
>>> y_g = np.linspace(-8, 4, 3)
>>> x2d, y2d = np.meshgrid(x_g, y_g)
>>> print x2d, "\n", y2d
```

```
[[ 0.  4.5  9. ]
 [ 0.  4.5  9. ]
 [ 0.  4.5  9. ]],
[[-8. -8. -8.]
 [-2. -2. -2.]
 [ 4.  4.  4.]]
```

X-values for each cell in a grid

Y-values for each cell in a grid

## Let's start doing some calculations with arrays

# General array operations: Method 1 - the OLD way

Multiply two arrays together, element-by-element:

```
import numpy as np
a = np.array([[2, 3.2, 5.5, -6.4],
          [3, 1, 2.1, 21]])
b = np.array([[4, 1.2, -4, 9.1],
          [6, 21, 1.5, -27]])
shape_a = np.shape(a)

product_ab = np.zeros(shape_a, dtype='f')

for i in xrange(shape_a[0]):
    for j in xrange(shape_a[1]):
        product_ab[i, j] = a[i, j] * b[i, j]
```

## General array operations: Method 1: the OLD way

- Note the use of xrange (which is like range, but provides only one element of the list at a time) to create a list of indices.
- Loops are relatively slow.
- You could also add a line to check that the two arrays have the same shape.

## General array operations: Method 2: array syntax

```
product_ab = a * b
```

It's a one liner!

```
c = a + b
```

- Operand shapes are automatically checked for compatibility.
- You do not need to know the rank of the arrays ahead of time, so the same line of code works on arrays of any dimension.
- This makes them *much faster* than loops.

## Testing inside an array: Method 1: the OLD way

Often, you will want to do calculations on an array that involves conditions. For example:

You have a 2-D array a and you want to return an array answer which is *double the value* when the element in a is greater than 5 and less than 10, and output zero when it is not.

Here's the code…

## Testing inside an array: Method 1: the OLD way

```
answer = np.zeros(np.shape(a), dtype='f')

for i in xrange(np.shape(a)[0]):
    for j in xrange(np.shape(a)[1]):
        if (a[i,j] > 5) and (a[i,j] < 10):
            answer[i,j] = a[i,j] * b[i,j]
        else:
            pass # i.e. do nothing
```

## Testing inside an array
## Method 2: array syntax

Comparison operators (implemented either as operators or functions) act element-wise, and return a Boolean array. For instance:

```
answer = a > 5
answer = np.greater(a, 5)
```

Boolean operators are implemented as functions that also act element-wise (e.g., `logical and`, `logical or`).

# Testing inside an array—Method 2 (array syntax) II

The `where` function tests any condition and applies operations for true and false cases, as specified, on an element-wise basis. For instance, consider the following case where you can assume `a = np.arange(10)`:

```
condition = np.logical_and(a > 5, a < 10)
answer = np.where(condition, a * 2, 0)
```

## Testing inside an array—Method 2 (array syntax) II

The above code implements the example we saw previously (i.e., say you have a 2-D array a and you want to return an array answer which is double the value when the element in a is greater than 5 and less than 10, and output zero when it is not) and is both cleaner and runs faster.

# Testing inside an array—Method 2 (array syntax) III

You can also accomplish what the where function does in the previous slide by taking advantage of how arithmetic operations on boolean arrays treat True as 1 and False as 0.

By using multiplication and addition, the boolean values become selectors. For instance:

## Testing inside an array—Method 2 (array syntax) III

```
condition = np.logical_and(a > 5, a < 10)
answer = ((a*2) * condition) +
            (0 * np.logical_not(condition))
```

- This method is also faster than loops.

## Additional array functions

- Basic mathematical functions: `np.sin`, `np.exp`, `np.interp`, etc.
- Basic statistical functions: `correlate`, `histogram`, `hamming`, `fft`, etc.
- NumPy has a lot of stuff! Use `dir(np)`, `help(np)`, as well as `help(np.x)`, where x is the name of a function, to get more information.

## Handling missing values (using masked arrays)

# Introducing a masked array

A **masked array** includes:

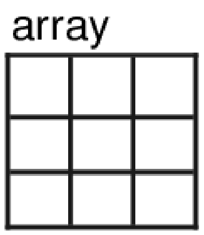
- a mask of *bad values* travels with the array.

Those elements deemed bad are treated as if they did not exist. Operations using the array automatically utilize the mask of bad values.

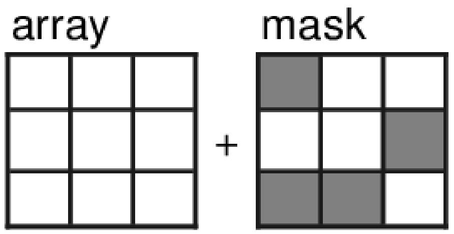
Typically bad values may represent something like a **land mask** (I.e. *sea surface temperature* only exists where there is ocean).

# Comparing arrays and masked arrays

**Arrays:**
**(numpy)**



**Masked Arrays:**
**(numpy.ma)**



## Constructing Masked Arrays

All functions are part of the `numpy.ma` submodule, and in these examples, assuming I import that submodule with `import numpy.ma as MA` and NumPy is imported as `import numpy as np`.

## Constructing Masked Arrays

Make a masked array by explicitly specifying a mask and fill value:

```
a = MA.masked_array(data=[1,2,3],
        mask=[True, True, False],
        fill_value=1e20)
```

# Working with Masked Arrays 1

Make a masked array by masking values greater than a value:

```
 a = MA.masked_greater([1,2,3,4], 3)
```

Make a masked array by masking values that meet a condition:

```
data = np.array([1,2,3,4,5])
a = MA.masked_where(np.logical_and
            (data>2, data<5), data)
```

## Working with Masked Arrays 2

Make a regular NumPy array from a masked array:

```
b = MA.masked_array(data=[1,2,3],
            mask=[True, True, False])
a = MA.filled(b)
```

NB: **Bad values** (i.e., missing values) have masked values set to `True` in a masked array.

Reference: The NumPy in-source documentation.

## Conclusions

- NumPy is a powerful array handling package that provides the array handling functionality of IDL, Matlab, Fortran 90 etc.
- Array syntax enables you to write more streamlined and flexible code: The same code can handle operations on arrays of arbitrary rank.
- Masked arrays extend the functionality by providing support for "bad values".