Numpy

Handling Arrays in Python

Thanks to all contributors:

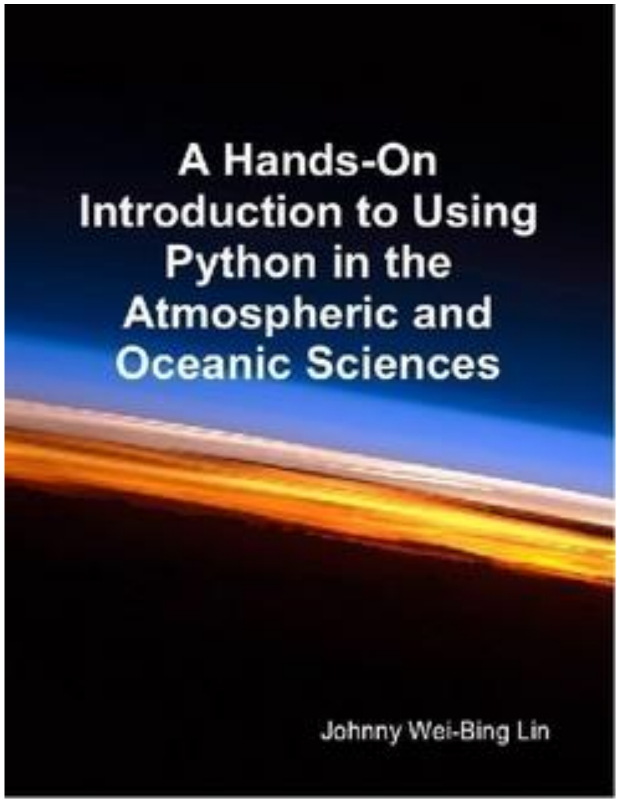
Alison Pamment, Sam Pepler, Ag Stephens, Stephen Pascoe, Anabelle Guillory, Graham Parton, Esther Conway, Wendy Garland, Alan Iwi, Matt Pritchard and Tommy Godfrey.

With special thanks to…

Johnny Lin wrote a great python/atmospheric science book with exercises, examples, presentations, books etc.

It is Python 2 however.

Much of this is borrowed from…


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

https://www.johnny-lin.com/pyintro/

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]])
```

- 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=np.int32)
```

Typecodes for arrays

Some common typecodes (which are strings):

`np.float64:`	`Double precision float`
`np.float32:`	`Single precision float`
`np.int8:`	`Byte`
`np.int64:`	`Long integer (64-bit)`

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=np.float64)
```

To create an 1-D array similar to `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[:]`

Multi-dimensional array indexing

For multi-dimensional arrays, indexing between different dimensions is separated by commas (fastest-varying dimension is last).

e.g. a[4, 2] ← [row, col]

Slicing rules also work as applied for each dimension.

e.g. a[1, :, 4:7]

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, :]   →  array([1, 22, 4, 0.1, 5.3, -9])
a[1, 1:4] →  array([22, 4, 0.1])
```

Interrogating arrays

- Numpy has many functions that give info about arrays.
- Examples for array "`a`" (assuming you imported numpy as `np`):
  - Shape: `np.shape(a)`
  - Rank (number of dimensions): `np.ndim(a)`
  - Number of elements: `np.size(a)` ← *don't use* *`len`*
  - Maximum: `np.max(a)` Similarly `np.min(a)`

Array manipulation

There are many functions to manipulate arrays, e.g.:

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)`

Arrays as objects

Arrays have methods or attributes, including equivalents of the more commonly used `np.` .... functions, e.g.:

```
a.shape       ⇔   np.shape(a)
a.max()       ⇔   np.max(a)
a.repeat(3)   ⇔   np.repeat(a,3)
```

as well as various others, e.g.:

`a.dtype` ← *interrogate data type*

`b = a.astype(np.int32)` ← *convert data type*

although much else exists only as `np.` .... , e.g.

`np.average(a)` ← ~~`a.average`~~ *doesn't exist*

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)
[[ 0.  4.5  9. ]
 [ 0.  4.5  9. ]
 [ 0.  4.5  9. ]]
>>> print(y2d)
[[-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

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]])
```

General array operations

```
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

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

You have a 2-D array and you want to *double each value* when the element is between 5 and 10, and set to zero when it is not.

Testing inside an array

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 provided which also act element-wise, e.g.:

```
np.logical_not(a > 3)
np.logical_and(a > 3, a < 5)
```

Testing inside an array

The `where` function tests any condition and applies operations for true and false cases, as specified, on an element-wise basis.

Our use case can be coded:

```
condition = np.logical_and(a > 5, a < 10)

answer = np.where(condition, a * 2, 0)
```

Testing inside an array

The code implements the example we saw previously:

- you have a 2-D array `a`
- you want to return an array `answer` which is:
  - double each value when the element is > 5 and < 10
  - zero when it is not

This is both clean and fast.

Additional array functions

- Basic mathematical functions: `np.sin`, `np.exp`, `np.interp`, etc.
- Basic statistical functions: `np.correlate`, `np.histogram`, `np.hamming`, `np.fft`, etc.
- NumPy has a lot of stuff! For more info, use:
  - `dir(np)` and `help(np)`
  - `help(np.x)`, where `x` is the name of a function
  - `dir(a)` and `help(a)`, where `a` is the name of an array

Handling missing values (using masked arrays)

Introducing a masked array

A **masked array** includes:

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

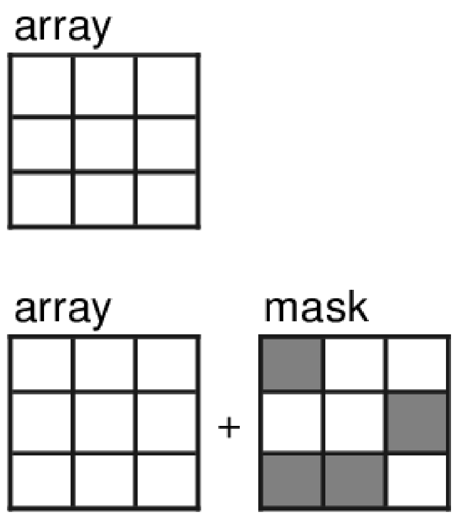
Those elements deemed bad are treated as if they did not exist. Operations using the array automatically utilise the mask of bad values.

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

Comparing arrays and masked arrays

Arrays:
(numpy)

Masked Arrays:
(numpy.ma)



Constructing Masked Arrays 1

All functions are part of the `numpy.ma` submodule.

In these examples, assume that I import that submodule with:

```
import numpy.ma as MA
```

and NumPy is imported as

```
import numpy as np
```

Constructing Masked Arrays 2

Make a masked array by explicitly specifying a mask (missing values have mask = True):

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

```
>>> a
masked_array(data = [-- -- 3],
mask = [ True  True False],
fill_value = 999999)
```

$\leftarrow$ *discussed later*

Constructing Masked Arrays 3

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

```
b = MA.masked_greater([1,2,3,4,5],3)
```

```
>>> b
masked_array(data = [1, 2, 3, --, --],
mask = [False, False, False, True, True],
fill_value = 999999)
```

Constructing Masked Arrays 4

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

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

```
>>> c
masked_array(data = [1, 2, --, --, 5],
mask = [False, False, True, True, False],
fill_value = 999999)
```

Masked arrays: example operations

(with **b** and **c** arrays as constructed in previous slide)

Masked when:
b > 3
c > 2, c < 5

Expression	Values including mask
`b`	`[ 1   2   3 --  --]`
`c`	`[ 1   2 -- --   5]`
`b * b`	`[ 1   4   9 --  --]`
`b + 1`	`[ 2   3   4 --  --]`
`b + c`	`[ 2   4 -- --  --]`

Masked arrays: fill values

An array can be "filled", which replaces masked values with the `fill_value`

```
>>> print(a.filled())      ← or MA.filled(a)
array([999999, 999999, 3]) ← numpy array
```

- Typical use: when writing to a file
- You should use a value outside the valid data range
- Can override default when creating an array, e.g.
  ```
  MA.masked_array(data=...,mask=...,
                         fill_value=1e30)
  ```

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".
- Other libraries, such as Pandas, netCDF4, cf-python, iris and Xarray all use NumPy.