Chapter 4

Array Operations

4.1 What is an array and the NumPy package

In Ch. 3, we were introduced to lists, which look a lot like Fortran arrays, except lists can hold values of any type. The computational overhead to support that flexibility, however, is non-trivial, and so lists are not practical to use for most scientific computing problems: lists are too slow. To solve this problem, Python has a package called NumPy[1] which defines an array data type that in many ways is like the array data type in Fortran, IDL, etc.

NumPy arrays are like lists except all elements are the same type.

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; and array operations are supported. To utilize NumPy's functions and attributes, you import the package `numpy`. Because NumPy functions are often used in scientific computing, you usually import NumPy as an alias, e.g., `import numpy as N`, to save yourself some typing (see p. 41 for more about importing as an alias). Note that in this chapter and the rest of the book, if you see the alias `N` in code without `import numpy as N` explicitly state, you can assume that `N` was defined by such an import statement somewhere earlier in the code.

Importing NumPy.

4.2 Creating arrays

The most basic way of creating an array is to take an existing list and convert it into an array using the `array` function in NumPy. Here is a basic example:

[1]There are other array packages for Python, but the community has now converged on NumPy.

Example 24 (Using the `array` function on a list):

Assume you have the following list:

```
mylist = N.array([[2, 3, -5],[21, -2, 1]])
```

then you can create an array `a` with:

```
import numpy as N
a = N.array(mylist)
```

Creating arrays using `array`.

The `array` function will match the array type to the contents of the list. Note that the elements of `mylist` have to be convertible to the same type. Thus, if the list elements are all numbers (floating point or integer), the `array` function will work fine. Otherwise, things could get dicey.

Making arrays of a given type.

Sometimes you will want to make sure your NumPy array elements are of a specific type. To force a certain numerical type for the array, set the `dtype` keyword to a type code:

Example 25 (Using the `dtype` keyword):

Assume you have a list `mylist` already defined. To make an array `a` from that list that is double-precision floating point, you'd type:

```
import numpy as N
a = N.array(mylist, dtype='d')
```

The `dtype` keyword and common array typecodes.

where the string `'d'` is the **typecode** for double-precision floating point. Some common typecodes (which are all strings) include:

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

Often you will want to create an array of a given size and **shape,** but you will not know in advance what the element values will be. To create an array of a given shape filled with zeros, use the `zeros` function, which takes the shape of the array (a tuple) as the single positional input argument (with `dtype` being optional, if you want to specify it):

Example 26 (Using the `zeros` function):

Let's make an array of zeros of shape (3,2), i.e., three rows and two columns in shape. Type in:

Using `zeros` to create a zero-filled array of a given shape.

```
import numpy as N
a = N.zeros((3,2), dtype='d')
```

Print out the array you made by typing in `print a`. Did you get what you expected?

Solution and discussion: You should have gotten:

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

Note that you don't have to type `import numpy as N` prior to every use of a function from NumPy, as long as earlier in your source code file you have done that import. In the examples in this chapter, I will periodically include this line to remind you that `N` is now an alias for the imported NumPy module. However, in your own code file, if you already have the `import numpy as N` statement near the beginning of your file, you do not have to type it in again as per the example. Likewise, if I do not tell you to type in the `import numpy as N` statement, and I ask you to use a NumPy function, I'm assuming you already have that statement earlier in your code file.

You only have to import NumPy once in your module file.

Also note that while the input shape into `zeros` is a tuple, which all array shapes are, if you type in a list, the function call will still work.

Another array you will commonly create is the array that corresponds to the output of `range`, that is, an array that starts at 0 and increments upwards by 1. NumPy provides the `arange` function for this purpose. The syntax is

The `arange` function.

the same as `range`, but it optionally accepts the `dtype` keyword parameter if you want to select a specific type for your array elements:

Example 27 (Using the `arange` function):

Let's make an array of 10 elements, starting from 0, going to 9, and incrementing by 1. Type in:

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

Print out the array you made by typing in `print a`. Did you get what you expected?

Solution and discussion: You should have gotten:

```
>>> print a
[0 1 2 3 4 5 6 7 8 9]
```

Be careful that `arange` gives you the array *type* you want.

Note that because the argument of `arange` is an integer, the resulting array has integer elements. If, instead, you had typed in `arange(10.0)`, the elements in the resulting array would have been floating point. You can accomplish the same effect by using the `dtype` keyword input parameter, of course, but I mention this because sometimes it can be a gotcha: you intend an integer array but accidentally pass in a floating point value for the number of elements in the array, or vice versa.

4.3 Array indexing

Array indices start with 0.

Like lists, element addresses start with zero, so the first element of a 1-D array `a` is `a[0]`, the second is `a[1]`, etc. Like lists, you can also reference elements starting from the end, e.g., element `a[-1]` is the last element in a 1-D array `a`.

Array slicing rules.

Array slicing follows rules very similar to list slicing:

- Element addresses in a range are separated by a colon.
- The lower limit is inclusive, and the upper limit is exclusive.
- If one of the limits is left out, the range is extended to the end of the range (e.g., if the lower limit is left out, the range extends to the very beginning of the array).

- Thus, to specify all elements, use a colon by itself.

Here's an example:

Example 28 (Array indexing and slicing):
Type the following in a Python interpreter:

```
a = N.array([2, 3.2, 5.5, -6.4, -2.2, 2.4])
```

What does `a[1]` equal? `a[1:4]`? `a[2:]`? Try to answer these first without using the interpreter. Confirm your answer by using `print`.

Solution and discussion: You should have gotten:

```
>>> print a[1]
3.2
>>> print a[1:4]
[ 3.2  5.5 -6.4]
>>> print a[2:]
[ 5.5 -6.4 -2.2  2.4]
```

For multi-dimensional arrays, indexing between different dimensions is separated by commas. Note that the fastest varying dimension is always the last index, the next fastest varying dimension is the next to last index, and so forth (this follows C convention).[2] 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). Here's an example:

Multi-dimensional array indexing and slicing.

Example 29 (Multidimensional array indexing and slicing):
Consider the following typed into a Python interpreter:

```
import numpy as N
a = N.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]])
```

[2] See http://docs.scipy.org/doc/numpy/reference/arrays.ndarray.html and the definition of "row-major" in http://docs.scipy.org/doc/numpy/glossary.html (both accessed August 9, 2012).

What is `a[1,2]` equal to? `a[:,3]`? `a[1,:]`? `a[1,1:4]`?

Solution and discussion: You should have obtained:

```
>>> print a[1,2]
4.0
>>> print a[:,3]
[ -6.4   0.1  21. ]
>>> print a[1,:]
[  1.   22.    4.    0.1   5.3  -9. ]
>>> print a[1,1:4]
[ 22.    4.    0.1]
```

Note that when I typed in the array I did not use the line continuation character at the end of each line because I was entering in a list, and by starting another line after I typed in a comma, Python automatically understood that I had not finished entering the list and continued reading the line for me.

4.4 Exercises in creating and indexing arrays

▷ **Exercise 12 (Creating an array of zeros):**
What is the code to create a 4 row, 5 column array of single-precision floating point zeros and assign it to the variable `a`?

Solution and discussion: The `zeros` function does the trick. Note that the first argument in the solution is a tuple that gives the shape of the output array, so the first argument needs the extra set of parentheses that says the sequence is a tuple:

```
a = N.zeros((4,5), dtype='f')
```

▷ **Exercise 13 (Using a multidimensional array):**
Consider the example array from Example 29, here repeated:

```
import numpy as N
a = N.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]])
```

1. What is `a[:,3]`?

2. What is `a[1:4,0:2]`? (Why are there no errors from this specification?)

3. What will `b = a[1:,2]` do? What will `b` be? Reason out first what will happen, then try it to see. If you were wrong, why were you wrong?

Solution and discussion: My answers:

1. `a[:,3]` is `[-6.4, 0.1, 21]`.

2. `a[1:4,0:2]`? selects the last two rows and first three columns as a subarray. There are no errors because while there is no "threeth" row, the row slicing works until it's out of rows.

3. `b` is the subarray consisting of the last two rows and the third column. The code assigns that subarray to the variable `b`.

4.5 Array inquiry

Some information about arrays comes through functions that act on arrays; other information comes through attributes attached to the array object. (Remember that basically everything in Python is an object, including arrays. In Section 7.4 we'll be talking more about array attributes.) Let's look at some array inquiry examples:

Example 30 (Array inquiry):

Import NumPy as the alias `N` and create a 2-D array `a`. Below are some array inquiry tasks and the Python code to conduct these tasks. Try these commands out in your interpreter and see if you get what you expect.

Finding the shape, rank, size, and type of an array.

- Return the shape of the array: `N.shape(a)`
- Return the **rank** of the array: `N.rank(a)`
- Return the number of elements in the array: `N.size(a)`
- Typecode of the array: `a.dtype.char`

Solution and discussion: Here are some results using the example array from Example 29:

```
>>> print N.shape(a)
(3, 6)
>>> print N.rank(a)
2
>>> print N.size(a)
18
>>> print a.dtype.char
d
```

Note that you should *not* use `len` for returning the number of elements in an array. Also, the `size` function returns the total number of elements in an array. Finally, `a.dtype.char` is an example of an array attribute; notice there are no parentheses at the end of the specification because an attribute variable is a piece of data, not a function that you call.

Use `size`, not `len`, for arrays.

Array inquiry enables you to write flexible code.

The neat thing about array inquiry functions (and attributes) is that you can write code to operate on an array in general instead of a specific array of given size, shape, etc. This allows you to write code that can be used on arrays of all types, with the exact array determined at run time.

4.6 Array manipulation

In addition to finding things about an array, NumPy includes many functions to manipulate arrays. Some, like `transpose`, come from linear algebra, but NumPy also includes a variety of array manipulation functions that enable you to massage arrays into the form you need to do the calculations you want. Here are a few examples:

Example 31 (Array manipulation):

Reshaping, transposing, and other array manipulation functions.

Import NumPy as the alias `N` and create one 6-element 1-D array `a`, one 8-element 1-D array `b`, and one 2-D array `c` (of any size and shape). Below are some array manipulation tasks and the Python code to conduct those tasks. Try these commands out in your interpreter and see if you get what you expect.

- Reshape the array and return the result, e.g.:

```
N.reshape(a,(2,3))
```

- Transpose the array and return the result:

```
N.transpose(c)
```

  (Note that I'm asking you to use `transpose` on the 2-D array; the transpose of a 1-D array is just the 1-D array.)

- Flatten the array into a 1-D array and return the result:

```
N.ravel(a)
```

- Concatenate arrays and return the result:

```
N.concatenate((a,b))
```

  Note that the function `concatenate` has *one* positional argument (not two, as the above may seem to suggest). That one argument is a tuple of the arrays to be concatenated. This is why the above code has "double" parenthesis.

- Repeat array elements and return the result, e.g.:

```
N.repeat(a,3)
```

- Convert array `a` to another type, e.g.:

Converting an array to another type.

```
d = a.astype('f')
```

  The argument of `astype` is the typecode for `d`. This is an example of an object method; we'll explain array object methods more in Section 7.4.

Solution and discussion: Here's my solution for arrays `a` and `b`, where `a = N.arange(6)` and `b = N.arange(8)`, and the 2-D array from Example 29 is now set to the variable `c`:

```
>>> print N.reshape(a,(2,3))
[[0 1 2]
 [3 4 5]]
>>> print N.transpose(c)
[[  2.    1.    3. ]
 [  3.2  22.    1. ]
 [  5.5   4.    2.1]
 [ -6.4   0.1  21. ]
 [ -2.2   5.3   1.1]
 [  2.4  -9.   -2. ]]
>>> print N.ravel(a)
[0 1 2 3 4 5]
>>> print N.concatenate((a,b))
[0 1 2 3 4 5 0 1 2 3 4 5 6 7]
>>> print N.repeat(a,3)
[0 0 0 1 1 1 2 2 2 3 3 3 4 4 4 5 5 5]
>>> d = a.astype('f')
>>> print d
[ 0.  1.  2.  3.  4.  5.]
```

You'll want to consult a NumPy reference (see Section 10.3) to get a full list of the array manipulation functions available, but here's one more snazzy function I wanted to mention. In the atmospheric and oceanic sciences, we often find ourselves using 2-D regularly gridded slices of data where the x- and y-locations of each array element is given by the corresponding elements of the x and y vectors. Wouldn't it be nice to get a 2-D array whose elements are the x-values for each column and a 2-D array whose elements are the y-values for each row? The `meshgrid` function does just that:

The `meshgrid` function.

Example 32 (The `meshgrid` function):

Consider the following code that creates two vectors, `lon` and `lat`, that hold longitude and latitude values (in degrees), respectively, and then assigns the result of `N.meshgrid(lon,lat)` to a variable `a`:

```
import numpy as N
lon = N.array([0, 45, 90, 135, 180, 225, 270, 315, 360])
lat = N.array([-90, -45, 0, 45, 90])
a = N.meshgrid(lon,lat)
```

What type is `a`? What is `a[0]`? `a[1]`?

Solution and discussion: The variable `a` is a tuple of two elements. The first element of `a`, i.e., `a[0]`, is a 2-D array:

```
>>> print a[0]
[[  0  45  90 135 180 225 270 315 360]
 [  0  45  90 135 180 225 270 315 360]
 [  0  45  90 135 180 225 270 315 360]
 [  0  45  90 135 180 225 270 315 360]
 [  0  45  90 135 180 225 270 315 360]]
```

and the second element of the tuple `a`, i.e., `a[1]` is also a 2-D array:

```
>>> print a[1]
[[-90 -90 -90 -90 -90 -90 -90 -90 -90]
 [-45 -45 -45 -45 -45 -45 -45 -45 -45]
 [  0   0   0   0   0   0   0   0   0]
 [ 45  45  45  45  45  45  45  45  45]
 [ 90  90  90  90  90  90  90  90  90]]
```

The columns of `a[0]` are the longitude values at each location of the 2-D grid whose longitude locations are defined by `lon` and whose latitude locations are defined by `lat`. The rows of `a[1]` are the latitude values at each location of the same 2-D grid (i.e., that grid whose longitude locations are defined by `lon` and whose latitude locations are defined by `lat`). Which is what we wanted ☺.

An aside: Note that the first row (i.e., the zeroth row) in `a[1]` is the first one printed, so going from top-to-bottom, you are moving in latitude values from south-to-north. Thus:

```
>>> print a[1][0,:]
[-90 -90 -90 -90 -90 -90 -90 -90 -90]
```

will print the -90 degrees latitude row in `a[1]`. Remember that 2-D arrays in NumPy are indexed [row, col], so the slicing syntax `[0,:]` will select all columns in the first row of a 2-D NumPy array.

4.7 General array operations

So far we've learned how to make arrays, ask arrays to tell us about themselves, and manipulate arrays. But what scientists really want to do with arrays is make calculations with them. In this section, we discuss two ways to do exactly that. Method 1 uses `for` loops, in analogue to the use of loops in traditional Fortran programming, to do element-wise array calculations. Method 2 uses array syntax, where looping over array elements happens implicitly (this syntax is also found in Fortran 90 and later versions, IDL, etc.).

4.7.1 General array operations: Method 1 (loops)

Using `for` loops to operate on arrays.

The tried-and-true method of doing arithmetic operations on arrays is to use loops to examine each array element one-by-one, do the operation, and then save the result in a results array. Here's an example:

Example 33 (Multiply two arrays, element-by-element, using loops):

Consider this code:

```
1  import numpy as N
2  a = N.array([[2, 3.2, 5.5, -6.4],
3               [3,   1, 2.1,   21]])
4  b = N.array([[4, 1.2,  -4,  9.1],
5               [6,  21, 1.5,  -27]])
6  shape_a = N.shape(a)
7  product_ab = N.zeros(shape_a, dtype='f')
8  for i in xrange(shape_a[0]):
9      for j in xrange(shape_a[1]):
10         product_ab[i,j] = a[i,j] * b[i,j]
```

Can you describe what is happening in each line? (We haven't talked about `xrange` yet, but take a guess as to what it does.)

Solution and discussion: In the first four lines after the `import` line (lines 2–5), I create arrays `a` and `b`. They are both two row, four column arrays. In the sixth line, I read the shape of array `a` and save it as the variable `shape_a`. Note that `shape_a` is the tuple (2,4). In the seventh line, I create a results array of the same shape of `a` and `b`, of single-precision floating point type, and with each element filled with zeros. In the last three lines (lines 8–10), I loop through all rows (the number of which is given by `shape_a[0]`) and all columns (the number of which is given by `shape_a[1]`), by index.

Thus, `i` and `j` are set to the element addresses for rows and columns, respectively, and line 10 does the multiplication operation and sets the product in the results array `product_ab` using the element addresses.

So, what is the `xrange` function? Recall that the `range` function provides an n-element list of the integers 0 to $n-1$, incremented by 1, and is useful for providing the element addresses for lists (and arrays). The `range` function creates the entire list in memory when it is called, but for the purposes of looping through list/array element addresses, we're not interested in being able to access all the addresses all the time; we only need the element address for the current loop iteration. That's what `xrange` does; it provides only one element of the array element addresses list at a time. This makes the loop more efficient.

The `xrange` function makes looping more efficient.

One other note: In this example, I make the assumption that the shape of `a` and the shape of `b` are the same, but I should instead add a check that this is actually the case. While a check using an `if` statement condition such as:

Do not use logical equality to check equality between sequences.

```
N.shape(a) != N.shape(b)
```

will work, because equality between sequences is true if all corresponding elements are equal,[3] things get tricky, fast, if you are interested in more complex logical comparisons and boolean operations for arrays. For instance, the logic that works for `!=` doesn't apply to built-in Python boolean operators such as `and`. We'll see later on in Section 4.8.2 how to do element-wise boolean operations on arrays.

So, why wouldn't you want to use the looping method for general array operations? In three and a half words: Loops are (relatively) s-l-o-w. Thus, if you can at all help it, it's better to use array syntax for general array operations: your code will be faster, more flexible, and easier to read and test.

Loops are slower than array syntax.

4.7.2 General array operations: Method 2 (array syntax)

The basic idea behind array syntax is that, much of the time, arrays interact with each other on a corresponding element basis, and so instead of requiring the user to write out the nested `for` loops explicitly, the loops and element-wise operations are done implicitly in the operator. That is to say, instead of writing this code (assume arrays `a` and `b` are 1-D arrays of the same size):

What is array syntax?

[3] See the "Built-in Types" entry in the online Python documentation at http://docs.python.org/library/stdtypes.html#sequence-types-str-unicode-list-tuple-bytearray-buffer-xrange (accessed March 26, 2012).

```
c = N.zeros(N.shape(a), dtype='f')
for i in xrange(N.size(a)):
    c[i] = a[i] * b[i]
```

array syntax means you can write this code:

```
c = a * b
```

Let's try this with a specific example using actual numbers:

Example 34 (Multiply two arrays, element-by-element, using array syntax):

Type the following in a file and run it using the Python interpreter:

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

What do you get when you print out `product_ab`?

Solution and discussion: You should get something like this:

```
>>> print product_ab
[[   8.       3.84  -22.     -58.24]
 [  18.      21.       3.15 -567.   ]]
```

Arithmetic operators act element-wise by default on NumPy arrays.

In this example, we see that arithmetic operators are automatically defined to act element-wise when operands are NumPy arrays or scalars. (Operators do have function equivalents in NumPy, e.g., `product`, `add`, etc., for the situations where you want to do the operation using function syntax.) Additionally, the output array `c` is automatically created on assignment; there is no need to initialize the output array using `zeros`.

Array syntax already checks compatibility.

There are three more key benefits of array syntax. First, operand shapes are automatically checked for compatibility, so there is no need to check for that explicitly. Second, you do not need to know the rank (i.e., whether it is 1-D, 2-D, etc.) of the arrays ahead of time, so the same line of code works on arrays of *any* rank. Finally, the array syntax formulation runs faster than the equivalent code using loops! Simpler, better, faster: pretty cool, eh? ☺

Let's try another array syntax example:

Example 35 (Another array syntax example):

Type the following in a Python interpreter:

```
import numpy as N
a = N.arange(10)
b = a * 2
c = a + b
d = c * 2.0
```

What results? Predict what you think a, b, and c will be, then print out those arrays to confirm whether you were right.

Solution and discussion: You should get something like this:

```
>>> print a
[0 1 2 3 4 5 6 7 8 9]
>>> print b
[ 0  2  4  6  8 10 12 14 16 18]
>>> print c
[ 0  3  6  9 12 15 18 21 24 27]
>>> print d
[  0.   6.  12.  18.  24.  30.  36.  42.  48.  54.]
```

Arrays a, b, and c are all integer arrays because the operands that created those arrays are all integers. Array d, however, is floating point because it was created by multiplying an integer array by a floating point scalar. Python automatically chooses the type of the new array to retain, as much as possible, the information found in the operands.

4.7.3 Exercise on general array operations

▷ **Exercise 14 (Calculate potential temperature from arrays of T and p):** Write a function that takes a 2-D array of pressures (p, in hPa) and a 2-D array of temperatures (T, in K) and returns the corresponding potential temperature, assuming a reference pressure (p_0) of 1000 hPa. Thus, the function's return value is an array of the same shape and type as the input arrays. Recall that potential temperature θ is given by:

$$\theta = T\left(\frac{p_0}{p}\right)^{\kappa}$$

where κ is the ratio of the gas constant of dry air to the specific heat of dry air at constant pressure and equals approximately 0.286.

Solution and discussion: I will give two different solutions: one using loops and the other using array syntax. Using loops, you get:

```
import numpy as N
def theta(p, T, p0=1000.0, kappa=0.286):
    shape_input = N.shape(p)
    output = N.zeros(shape_input, dtype='f')
    for i in xrange(shape_input[0]):
        for j in xrange(shape_input[1]):
            output[i,j] = T[i,j] * (p0 / p[i,j])**(kappa)
    return output
```

Remember to use `return` when passing a result out of a function.

Note the use of keyword input parameters to provide potentially adjustable constants. Remember, to return anything from a function, you have to use the `return` command.

Using array syntax, the solution is even terser:

```
import numpy as N
def theta(p, T, p0=1000.0, kappa=0.286):
    return T * (p0 / p)**(kappa)
```

and the array syntax solution works for arrays of any rank, not just 2-D arrays.

An aside on documenting code: Python has a robust set of standardized ways to generate code documentation. The most basic construct, as you might guess, is the humble but ever-important comment line. The pound sign ("#") is Python's comment character, and all text after that symbol is ignored by the interpreter.

Python's comment character.

The most basic, specialized, built-in construct for documenting code is the **docstring.** These are strings set in triple quotes that come right after a `def` statement in a function. Here is my array syntax solution to Exercise 14 with a docstring added:

Documenting with the docstring.

```
import numpy as N
def theta(p, T, p0=1000.0, kappa=0.286):
    """Calculate the potential temperature.

    Returns a NumPy array of potential temperature that is
    the same size and shape as the input parameters.  The
    reference pressure is given by p0 and kappa is the
    ratio of the gas constant for dry air to the specific
    heat of dry air at constant pressure.

    Input parameters:
        :p:  Pressure [hPa].  NumPy array of any rank.
        :T:  Temperature [K].  NumPy array of any rank.
    """
    return T * (p0 / p)**(kappa)
```

Finally, there are a number of document generation packages that automatically convert Python code and code docstrings into web documentation. In the docstring example I give above, I use some reStructuredText conventions that will be nicely typeset by the Sphinx documentation generator. See http://docutils.sf.net/rst.html and http://sphinx.pocoo.org for details.

The Sphinx documentation generation package.

4.8 Testing inside an array

Often times, you will want to do calculations on an array that involves conditionals. For instance, you might want to loop through an array of data and check if any values are negative; if any exist, you may wish to set those elements to zero. To accomplish the first part of that task, you need to do some kind of testing while going through an array.

In Python, there are a few ways of doing this. The first is to implement this in a loop. A second way is to use array syntax and take advantage of comparison operators and specialized NumPy search functions.

4.8.1 Testing inside an array: Method 1 (loops)

In this method, you apply a standard conditional (e.g., `if` statement) while inside the nested `for` loops running through the array. This is similar to traditional Fortran syntax. Here's is an example:

Example 36 (Using looping to test inside an array):

Say you have a 2-D array a and you want to return an array answer which is double the value of the corresponding element in a when the element is greater than 5 and less than 10, and zero when the value of that element in a is not. What's the code for this task?

Solution and discussion: Here's the code:

```
answer = N.zeros(N.shape(a), dtype='f')
for i in xrange(N.shape(a)[0]):
    for j in xrange(N.shape(a)[1]):
        if (a[i,j] > 5) and (a[i,j] < 10):
            answer[i,j] = a[i,j] * 2.0
        else:
            pass
```

The pass command in blocks that do nothing.

The pass command is used when you have a block statement (e.g., a block if statement, etc.) where you want the interpreter to do nothing. In this case, because answer is filled with all zeros on initialization, if the if test condition returns False, we want that element of answer to be zero. But, all elements of answer start out as zero, so the else block has nothing to do; thus, we pass.

Again, while this code works, loops are slow, and the if statement makes it even slower. The nested for loops also mean that this code will only work for a 2-D version of the array a.

4.8.2 Testing inside an array: Method 2 (array syntax)

Is there a way we can do testing inside an array while using array syntax? That way, we can get the benefits of simpler code, the flexibility of code that works on arrays of any rank, and speed. The answer is, yes! Because NumPy has comparison and boolean operators that act element-wise and array inquiry and selection functions, we can write a variety of ways of testing and selecting inside an array while using array syntax. Before we discuss some of those ways, we need some context about using NumPy comparison operators and boolean array functions.

NumPy comparison operators and boolean array functions

NumPy has defined the standard comparison operators in Python (e.g., ==, <) to work element-wise with arrays. Thus, if you run these lines of code:

```
import numpy as N
a = N.arange(6)
print a > 3
```

the following array is printed out to the screen:

```
[False False False False True True]
```

Each element of the array a that was greater than 3 has its corresponding element in the output set to True while all other elements are set to False. You can achieve the same result by using the corresponding NumPy function greater. Thus:

```
print N.greater(a, 3)
```

Using comparison operators on arrays generate boolean arrays.

gives you the same thing. Other comparison functions are similarly defined for the other standard comparison operators; those functions also act element-wise on NumPy arrays.

Once you have arrays of booleans, you can operate on them using boolean operator NumPy functions. You cannot use Python's built-in and, or, etc. operators; those will not act element-wise. Instead, use the NumPy functions logical_and, logical_or, etc. Thus, if we have this code:

Must use NumPy functions to do boolean operations on arrays.

```
a = N.arange(6)
print N.logical_and(a>1, a<=3)
```

the following array will be printed to screen:

```
[False False True True False False]
```

The logical_and function takes two boolean arrays and does an element-wise boolean "and" operation on them and returns a boolean array of the same size and shape filled with the results.

With this background on comparison operators and boolean functions for NumPy arrays, we can talk about ways of doing testing and selecting in arrays while using array syntax. Here are two methods: using the where function and using arithmetic operations on boolean arrays.

The `where` function

IDL users will find this function familiar. The Python version of `where`, however, can be used in two ways: To directly select corresponding values from another array (or scalar), depending on whether a condition is true, and to return a list of array element indices for which a condition is true (which then can be used to select the corresponding values by selection with indices).

The syntax for using `where` to directly select corresponding values is the following:

`N.where(`*<condition>*`,` *<value if true>*`,` *<value if false>*`)`

Using `where` to get values when a condition is true.

If an element of *<condition>* is `True`, the corresponding element of *<value if true>* is used in the array returned by the function, while the corresponding element of *<value if false>* is used if *<condition>* is `False`. The `where` function returns an array of the same size and shape as *<condition>* (which is an array of boolean elements). Here is an example to work through:

Example 37 (Using `where` to directly select corresponding values from another array or scalar):

Consider the following case:

```
import numpy as N
a = N.arange(8)
condition = N.logical_and(a>3, a<6)
answer = N.where(condition, a*2, 0)
```

What is `condition`? `answer`? What does the code do?

Solution and discussion: You should get:

```
>>> print a
[0 1 2 3 4 5 6 7]
>>> print condition
[False False False False  True  True False False]
>>> print answer
[ 0  0  0  0  8 10  0  0]
```

The array `condition` shows which elements of the array `a` are greater than 3 and less than 6. The `where` call takes every element of array `a` where that is true and doubles the corresponding value of `a`; elsewhere, the output element from `where` is set to 0.

The second way of using `where` is to return a tuple of array element indices for which a condition is true, which then can be used to select the corresponding values by selection with indices. (This is like the behavior of IDL's `WHERE` function.) For 1-D arrays, the tuple is a one-element tuple whose value is an array listing the indices where the condition is true. For 2-D arrays, the tuple is a two-element tuple whose first value is an array listing the row index where the condition is true and the second value is an array listing the column index where the condition is true. In terms of syntax, you tell `where` to return indices instead of an array of selected values by calling `where` with only a single argument, the <*condition*> array. To select those elements in an array, pass in the tuple as the argument inside the square brackets (i.e., `[]`) when you are selecting elements. Here is an example:

Using `where` to get the indices where a condition is true.

Example 38 (Using `where` to return a list of indices):

Consider the following case:

```
import numpy as N
a = N.arange(8)
condition = N.logical_and(a>3, a<6)
answer_indices = N.where(condition)
answer = (a*2)[answer_indices]
```

What is `condition`? `answer_indices`? `answer`? What does the code do?

Solution and discussion: You should have obtained similar results as Example 37, except the zero elements are absent in `answer` and now you also have a tuple of the indices where `condition` is true:

```
>>> print a
[0 1 2 3 4 5 6 7]
>>> print condition
[False False False False  True  True False False]
>>> print answer_indices
(array([4, 5]),)
>>> print answer
[ 8 10]
```

The array `condition` shows which elements of the array `a` are greater than 3 and less than 6. The `where` call returns the indices where `condition` is true, and since `condition` is 1-D, there is only one element in the tuple `answer_indices`. The last line multiplies array `a` by two (which is also an array) and selects the elements from that array with addresses given by `answer_indices`.

Using `where` to obtain indices will return a 1-D array.

Note that selection with `answer_indices` will give you a 1-D array, even if `condition` is not 1-D. Let's turn array `a` into a 3-D array, do everything else the same, and see what happens:

```
import numpy as N
a = N.reshape( N.arange(8), (2,2,2) )
condition = N.logical_and(a>3, a<6)
answer_indices = N.where(condition)
answer = (a*2)[answer_indices]
```

The result now is:

```
>>> print a
[[[0 1]
  [2 3]]

 [[4 5]
  [6 7]]]
>>> print condition
[[[False False]
  [False False]]

 [[ True  True]
  [False False]]]
>>> print answer_indices
(array([1, 1]), array([0, 0]), array([0, 1]))
>>> print answer
[ 8 10]
```

Note how `condition` is 3-D and the `answer_indices` tuple now has three elements (for the three dimensions of `condition`), but `answer` is again 1-D.

Arithmetic operations using boolean arrays

You can also accomplish much of what the `where` function does in terms of testing and selecting by taking advantage of the fact that arithmetic operations on boolean arrays treat `True` as 1 and `False` as 0. By using multiplication and addition, the boolean values become selectors, because any value multiplied by 1 or added to 0 is that value. Let's see an example of how these properties can be used for selection:

Example 39 (Using arithmetic operators on boolean arrays as selectors):
Consider the following case:

```
import numpy as N
a = N.arange(8)
condition = N.logical_and(a>3, a<6)
answer = ((a*2)*condition) + \
         (0*N.logical_not(condition))
```

Solution and discussion: The solution is the same as Example 37:

```
>>> print a
[0 1 2 3 4 5 6 7]
>>> print condition
[False False False False  True  True False False]
>>> print answer
[ 0  0  0  0  8 10  0  0]
```

But how does this code produce this solution? Let's go through it step-by-step. The `condition` line is the same as in Example 37, so we won't say more about that. But what about the `answer` line? First, we multiply array `a` by two and then multiply that by `condition`. Every element that is `True` in `condition` will then equal double of `a`, but every element that is `False` in `condition` will equal zero. We then add that to zero times the `logical_not` of `condition`, which is `condition` but with all `True`s as `False`s, and vice versa. Again, any value that multiplies by `True` will be that value and any value that multiplies by `False` will be zero. Because `condition` and its "logical not" are mutually exclusive—if one is true the other is false—the sum of the two terms to create `answer` will select either `a*2` or `0`. (Of course, the array generated by `0*N.logical_not(condition)` is an array of zeros, but you can see how multiplying by something besides `0` will give you a different replacement value.)

Using arithmetic with boolean arrays as conditional selectors.

Also, note the continuation line character is a backslash at the end of the line (as seen in the line that assigns `answer`).

This method of testing inside arrays using arithmetic operations on boolean arrays is also faster than loops.

A simple way of seeing how fast your code runs.

An aside on a simple way to do timings: The time module has a function `time` that returns the current system time relative to the Epoch (a date that is operating system dependent). If you save the current time as a variable before and after you execute your function/code, the difference is the time it took to run your function/code.

Example 40 (Using `time` to do timings):

Type in the following and run it in a Python interpreter:

```
import time
begin_time = time.time()
for i in xrange(1000000L):
    a = 2*3
print time.time() - begin_time
```

What does the number that is printed out represent?

Solution and discussion: The code prints out the amount of time (in seconds) it takes to multiply two times three and assign the product to the variable `a` one million times. (Of course, it also includes the time to do the looping, which in this simple case probably is a substantial fraction of the total time of execution.)

4.8.3 Exercise on testing inside an array

▷ **Exercise 15 (Calculating wind speed from u and v):**

Write a function that takes two 2-D arrays—an array of horizontal, zonal (east-west) wind components (u, in m/s) and an array of horizontal, meridional (north-south) wind components (v, in m/s)—and returns a 2-D array of the magnitudes of the total wind, if the wind is over a minimum magnitude, and the minimum magnitude value otherwise. (We might presume that in this particular domain only winds above some minimum constitute "good" data while those below the minimum are indistinguishable from the minimum due to noise or should be considered equal to the minimum in order to properly represent the effects of some quantity like friction.)

Thus, your input will be arrays `u` and `v`, as well as the minimum magnitude value. The function's return value is an array of the same shape and type as the input arrays.

Solution and discussion: I provide two solutions, one using loops and one using array syntax. Here's the solution using loops:

```
import numpy as N
def good_magnitudes(u, v, minmag=0.1):
    shape_input = N.shape(u)
    output = N.zeros(shape_input, dtype=u.dtype.char)
    for i in xrange(shape_input[0]):
        for j in xrange(shape_input[1]):
            mag = ((u[i,j]**2) + (v[i,j]**2))**0.5
            if mag > minmag:
                output[i,j] = mag
            else:
                output[i,j] = minmag
    return output
```

Here's the solution using array syntax, which is terser and works with arrays of all ranks:

```
import numpy as N
def good_magnitudes(u, v, minmag=0.1):
    mag = ((u**2) + (v**2))**0.5
    output = N.where(mag > minmag, mag, minmag)
    return output
```

4.9 Additional array functions

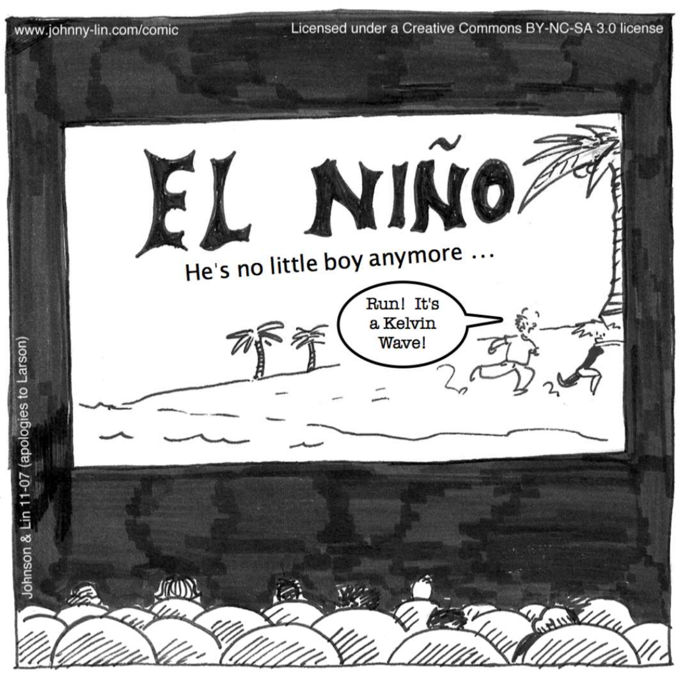
NumPy has many array functions, which include basic mathematical functions (`sin`, `exp`, `interp`, etc.) and basic statistical functions (`correlate`, `histogram`, `hamming`, `fft`, etc.). For more complete lists of array functions, see Section 10.3 for places to look. From the Python interpreter, you

See other listings for more array functions.

can also use `help(numpy)` as well as `help(numpy.`*x*`)`, where *x* is the name of a function, to get more information.

4.10 Summary

In this chapter, we saw that NumPy is a powerful array handling package that provides the array handling functionality of IDL, Matlab, Fortran 90, etc. We learned how to use arrays using the traditional Fortran-like method of nested `for` loops, but we also saw how array syntax enables you to write more streamlined and flexible code: The same code can handle operations on arrays of arbitrary rank. With NumPy, Python can be used for all of the traditional data analysis calculation tasks commonly done in the atmospheric and oceanic sciences. Not bad, for something that's free ☺.



Climatology Horror Films