ESCI 386 – IDL Programming for Advanced Earth Sciences Applications
Lesson 4 – Arrays

Reading: An Introduction to Programming with IDL, Chapter 7

GENERAL

• IDL is very adept at array operations.
  o This is because IDL is often used for image analysis, and images are efficiently
    stored and manipulated as arrays.
• Arrays can be of any data type (floating-point, integer, string).
  o All elements must be of the same data type. Can’t mix data types within an array.
• Elements are referenced by subscripts, or indices.
  o Use square brackets for referencing arrays.
• Array indices (subscripts) must start at 0.
• IDL has a lot of array operations and functions available. We will only scratch the
  surface in this lesson.
  o To learn more, use the online help to look up “arrays” and experiment with
    different functions.

CREATING ARRAYS

• Arrays can be created in several ways.
• One way is to use the appropriate array creating function, such as fltarr(), lonarr(), strarr().
  o Example: a = fltarr(5) would create a 5-element array named “a”.
• Another way to create an array is assigning values to it using square brackets.
  o Example: a = [23.0, 14.7, 37.0] would create a 3-element, floating point
  o Using this method, the array does not have to be first created with the fltarr
    function.
• Yet another way of creating an array is with a array generation functions such as findgen() and lindgen() functions, which create index arrays, which are arrays of
  evenly spaced whole numbers, beginning with 0.
  o findgen(3) => [0.0, 1.0, 2.0]
  o lindgen(3) => [0, 1, 2]
MULTI-DIMENSIONAL ARRAYS

- Multi-dimensional arrays are created in the same manner as one-dimensional arrays. The examples below demonstrate:
  - `a = fltarr(3, 4)` creates a 3 x 4 floating-point array.
  - `a = [[3, 4, 2], [2, 6, 9], [-1, 2, -6], [1, 6, 3]]` creates a 3 x 4 integer (or long) array filled with the values shown.
  - `a = lindgen(3, 4)` creates a long-integer index array whose values run from 0 to 11.

- IDL treats 2-D arrays as a matrix where the first index is the column, and the second index is the row.
- IDL uses column-major format (the same as Fortran), so that in intrinsic loops the first index (column index) varies more quickly than the second (row) index.

ACCESSING ARRAY ELEMENTS

- The examples below all refer to an array created using `a = lindgen(3, 3)`, so that `a` has a structure like

  \[
  \begin{array}{ccc}
  0 & 1 & 2 \\
  3 & 4 & 5 \\
  6 & 7 & 8 \\
  \end{array}
  \]

- Array elements are accessed by enclosing the index within square brackets.
  - `print, a[0,0] => 0`
  - `print, a[1,0] => 1`
  - `print, a[0,1] => 3`

- Multi-dimensional arrays can be accessed as though they are one-dimensional.
  - `print, a[0] => 0`
  - `print, a[1] => 1`
  - `print, a[3] => 3`

- Partial pieces of an array can be accessed using ranges and asterisks.
  - `print, a[2, 1:2] => 5 8`
• **Note that** `print, a[*,1]` => 3  4  5 
`print, a[*], 0:1]` => 0 1 2 
\[3 4 5\]

• **Variables can be used as array indices.**
  
  \[n = 3\]
  \[print, a[n] => 3\]

• **Arrays can be used as array indices!**
  
  \[a = [ 3, 6, 5, 1, 2, 9 ]\]
  \[b = [1, 3, 4 ]\]
  \[print, a[b] => [ 6, 1, 2 ]\]
  
  • **Note that this returned** `a[1]`, `a[3]`, and `a[4]` 

• **IDL even accepts floating-point values for array indices, by first converting them to integers (I wouldn’t make a habit of doing this, though).**
  
  \[print, a[2.8] => 2\]
  \[print, a[3.4] => 3\]
  \[print, a[2.5, 1.5] => 5\]

**INCREASING AND DECREASING THE SIZE OF AN EXISTING ARRAY**

• **The size of an existing array can be increased as follows**
  
  \[a = [ 1, 2, 3 ]\]
  \[a = [a, 4, 5 ]\]
  \[print, a => [ 1, 2, 3, 4, 5 ]\]
  \[a = [ -1, 0, a, 6, 7 ]\]
  \[print, a => [ -1, 0, 1, 2, 3, 4, 5, 6, 7 ]\]

• **The size of an existing array can be decreased as follows**
  
  \[a = [ -1, 0, 1, 2, 3, 4, 5, 6, 7 ]\]
  \[a = a[2:5]\]
  \[print, a => [ 1, 2, 3, 4 ]\]
SCALAR OPERATIONS ON ARRAYS

- A scalar value can be added, subtracted, multiplied, or divided through an entire array very easily.
  
  \[ a = \text{lindgen}(3,3) \]

  \[
  0 \ 1 \ 2 \\
  3 \ 4 \ 5 \\
  6 \ 7 \ 8 
  \]

  \[
  0 \ 2 \ 4 \\
  6 \ 8 \ 10 \\
  12 \ 14 \ 16 
  \]

  \[
  2 \ 3 \ 4 \\
  5 \ 6 \ 7 \\
  8 \ 9 \ 10 
  \]

  \[
  0 \ 0 \ 1 \\
  1 \ 2 \ 2 \\
  3 \ 3 \ 4 
  \]

ADDING, SUBTRACTING, DIVIDING, AND MULTIPLYING ENTIRE ARRAYS

- Entire arrays can be added, subtracted, multiplied, and divided by each other.
- IDL performs these operations by treating the arrays both as 1-dimensional arrays, and adding element by element.
- If the arrays are of the same size, the result is returned as an array of the same shape and size as the first array in the expression.

  \[ a = \text{lindgen}(2,3) \]

  \[ b = \text{lindgen}(3,2) \]

  \[
  0 \ 1 \\
  2 \ 3 \\
  4 \ 5 
  \]
• If one array is larger than the other, the result has the same shape and size as the smaller array.

```plaintext
a = lindgen(2,3)
b = lindgen(3,3)
```

```plaintext
print, a =>
0 1 2
 3 4 5

print, b =>
0 1 2
 3 4 5
 6 7 8

print, a + b =>
0 2
 4 6
 8 10
```

```plaintext
print, b + a =>
0 2 4
 6 8 10
```

```plaintext
print, a*b =>
0 1
 4 9
16 25
```

```plaintext
print, b*a =>
0 1 4
 9 16 25
```
print, b + a => 4 6
8 10

- You should be VERY CAREFUL about doing arithmetic on arrays of different shapes and sizes. If you do it, make sure you understand your results.

ROTATING AND TRANSPOSING ARRAYS
- Arrays can be rotated by multiples of 90° (and/or transposed) using the rotate() function.
  - The rotate() function accepts an array and a direction as the arguments.
    - The direction is a number from 0 to 7.
    - Directions of 0, 1, 2, and 3 rotate by 0°, 90°, 180°, and 270° respectively.
    - Directions of 4, 5, 6, and 7 first transpose the array, and then rotate by 0°, 90°, 180°, and 270° respectively.

  a = lingen(3,2)
  print, a =>
  0 1 2
  3 4 5

  print, rotate(a, 1) =>
  3 0
  4 1
  5 2

  print, rotate(a, 5) =>
  2 1 0
  5 4 3

- The transpose() function transposes an array without rotating it.
  - transpose(a) performs the exact same operation as rotate(a, 4)

SORTING ARRAYS
- An array can be sorted using the sort() function. The sort() function doesn’t return the sorted array; it returns a new array containing the indices of the original array in ascending order. For example:
  a = [ 3, 1, 6, 5, 8, 9 ]
print, sort(a) => [ 1, 0, 3, 2, 4, 5 ]

• So, to sort an array you can using the new array returned by `SORT()` as indices in the original array:
  
  a = [ 3, 1, 6, 5, 8, 9 ]
  b = sort(a)
  print, a[b] => [ 1, 3, 5, 6, 8, 9 ]

• The array could be sorted using one line of code as
  
  print, a[sort(a)]

SHIFTING ARRAYS

• The elements of an array can be shifted using the `SHIFT()` function.

• Positive shifts are to the right (or down), and negative shifts are to the left (or up).
  
  a = [ 0, 1, 2, 3, 4, 5]
  print, shift(a, 2) => [ 4, 5, 0, 1, 2, 3 ]
  print, shift(a, -2) => [ 2, 3, 4, 5, 0, 1 ]

• For multidimensional arrays you can shift each dimension.
  
  o To shift every element 2 spaces to the right and 3 spaces up, you would do as follows:
    
    1 2 3 4
    5 6 7 8
    a =
    9 10 11 12
    13 14 15 16
    
    15 16 13 14
    7 8 5 6
    print, shift(a, 2, -3) =>
    3 4 1 2
    11 12 9 10

RESAMPLING ARRAYS

• The `REBIN` function makes a larger array smaller by either sampling or averaging the elements.

  o The size of the new array must be an integer divisor of the size of the original array.
  
    a = findgen(6) => [0.0, 1.0, 2.0, 3.0, 4.0, 5.0]
print, rebin(a, 3) => [ 0.5, 2.5, 4.5 ]
print, rebin(a, 3, /sample) => [ 0.0, 2.0, 4.0 ]
print, rebin(a, 2) => [ 1.0, 4.0 ]
print, rebin(a, 2, /sample) => [ 0.0, 3.0 ]
print, rebin(a, 4) => Can’t do this one. 4 is not an integer divisor of 6.

- **REBIN** can be used on multiple dimensional arrays also.
- **REBIN** is handy for image processing, such as changing a higher resolution image to a lower resolution.

**REFORMING ARRAYS**

- You can reform an array (changing its shape) while keeping its original size using the **REFORM()** function.
  
  \[
  \begin{array}{cccc}
  1 & 2 & 3 & 4 \\
  5 & 6 & 7 & 8 \\
  \end{array}
  \]

  \[
  \begin{array}{ccc}
  1 & 2 \\
  3 & 4 \\
  5 & 6 \\
  7 & 8 \\
  \end{array}
  \]

  \[
  b = \text{reform}(a, 2, 4) \Rightarrow b=
  \]

**FILLING AN ARRAY WITH A CONSTANT VALUE**

- You can create a new array with all elements having the same value using the **REPLICATE()** function.

  \[
  a = \text{replicate}(5, 3, 2) \Rightarrow a=\frac{5}{5} \frac{5}{5} \frac{5}{5}
  \]

- For an array that already exists you can replace all its values with the **REPLICATE_INPLACE** procedure

  \[
  \text{replicate}_\text{inplace}, a, -2 \Rightarrow a=\frac{-2}{-2} \frac{-2}{-2} \frac{-2}{-2}
  \]

**WHERE FUNCTION**
• An extremely useful IDL function is the WHERE() function, which returns an array of indexes corresponding to all elements of an array where a certain expression is true.

• For example:
  
a = [3, -4, -6, 8, 12, 17, -100]
  
d = where(a ge 0)
  
print, d => [0, 3, 4, 5]
  
e = a[d]
  
print, e => [3, 8, 12, 17]
  
print, where((a ge 7) and (a le 15)) => [3, 4]

OTHER USEFUL ARRAY FUNCTIONS

• N_ELEMENTS() finds the number of elements of an array

• MAX() finds the maximum value of an array (can also return index of maximum value).
  
  o CAUTION: The MAX() function in IDL behaves very differently than that from FORTRAN!
    
      ▪ In IDL, the argument must be an array!
      
      ▪ So, if you want to find the maximum of two integers such as 3 and 4 in IDL, you cannot use MAX(3, 4) like you would in FORTRAN. You have to instead write it as MAX([3, 4]) so there is an array argument to the function.

• MIN() finds the minimum value of an array (can also return index of minimum value).

• REVERSE() reverses the elements of an array.

• SIZE() returns information about the number of dimensions and number of elements of an array.