Exercise 1: Introduction to NumPy arrays

Aim: Introduce basic NumPy array creation and indexing

Issues covered:

- Importing NumPy
- Creating an array from a list
- Creating arrays of zeros or ones
- Understanding array typecodes
- Array indexing and slicing

1. Let's create a numpy array from a list.

a. Import the "`numpy`" library as "`np`".
b. Create a list with values 1 to 10 and assign it to the variable "`x`".
c. Create an integer array from "`x`" and assign it to the variable "`a1`".
d. Create an array of floats from "`x`" and assign it to the variable "`a2`".
e. Print the data type of each array (use the attribute "`dtype`").

2. Let's create arrays in different ways.

a. Create an array of shape (2, 3, 4) of zeros.
b. Create an array of shape (2, 3, 4) of ones.
c. Create an array with values `0` to `999` using the "`np.arange`" function.

3. Let's look at indexing and slicing arrays.

a. Create an array from the list `[2, 3.2, 5.5, -6.4, -2.2, 2.4]` and assign it to the variable "`a`".
b. Do you know what `a[1]` will equal? Print it to see.
c. Try printing `a[1:4]` to see what that equals
d. Create a 2-D array from the following list and assign it to the variable "`a`":

```
[[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]]
```

e. Can you guess what the following slices are equal to? Print them to check your understanding.

```
a[:, 3]
a[1:4, 0:4]
a[1:, 2]
```

Solution 1: Introduction to NumPy arrays

1.

```
>>> import numpy as np
>>> x = range(1, 11)
>>> a1 = np.array(x, np.int32)
>>> a2 = np.array(x, np.float32)
>>> print(a1.dtype)
int32
>>> print(a2.dtype)
float32
```

2.

```
>>> arr = np.zeros((2, 3, 4))
>>> arr = np.ones((2, 3, 4))
>>> arr = np.arange(1000)
```

3.

```
>>> a = np.array([2, 3.2, 5.5, -6.4, -2.2, 2.4])
>>> print(a[1])
3.2
>>> print(a[1:4])
[ 3.2  5.5 -6.4]
>>> 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]])
>>> print(a[:, 3])
[ -6.4   0.1  21. ]
>>> print(a[1:4, 0:4])
[[  1.   22.    4.    0.1]
 [  3.    1.    2.1  21. ]]
>>> print(a[1:, 2])
[ 4.   2.1]
```

Exercise 2: Interrogating and manipulating arrays

Aim: Learn how to interrogate and manipulate NumPy arrays

Issues covered:

- Interrogating the properties of an array
- Manipulating arrays to change their properties

1. Let's interrogate an array to find out its characteristics.

a. Create a 2-D array of shape (2, 4) containing two lists (`range(4), range(10, 14))` and assign it to the variable "`arr`".
b. Print the shape of the array.
c. Print the size of the array.
d. Print the maximum and minimum of the array.

2. Let's generate new arrays by modifying our array.

a. Continue to use the array "`arr`" as defined above.
b. Print the array re-shaped to (2, 2, 2).
c. Print the array transposed.
d. Print the array flattened to a single dimension.
e. Print the array converted to floats.

Solution 2: Interrogating and manipulating arrays

1.

```
>>> import numpy as np
>>> arr = np.array([range(4), range(10, 14)])
>>> print(arr.shape)
(2, 4)
>>> print(arr.size)
8
>>> print(arr.max())
13
>>> print(arr.min())
0
```

2.

```
>>> print(np.reshape(arr, (2, 2, 2)))
[[[ 0  1]
  [ 2  3]]

 [[10 11]
  [12 13]]]
>>> print(np.transpose(arr))
[[ 0 10]
 [ 1 11]
 [ 2 12]
 [ 3 13]]
>>> print(np.ravel(arr))
[ 0  1  2  3 10 11 12 13]
>>> print(arr.astype(np.float64))
[[  0.   1.   2.   3.]
 [ 10.  11.  12.  13.]]
```

Exercise 3: Array calculations and operations

Aim: Use NumPy arrays in mathematical calculations

Issues covered:

- Mathematical operations with arrays
- Mathematical operations mixing scalars and arrays
- Comparison operators and Boolean operations on arrays
- Using the "where" method
- Writing a function to work on arrays

1. Let's perform some array calculations.

a. Create a 2-D array of shape (2, 4) containing two lists (`range(4), range(10, 14)`) and assign it to the variable `"a"`.
b. Create an array from a list `[2, -1, 1, 0]` and assign it to the variable `"b"`.
c. Multiply array `"a"` by `"b"` and view the result. Do you understand how NumPy has used its *broadcasting* feature to do the calculation even though the arrays are different shapes?
d. Multiply array `"b"` by `100` and assign the result to the variable `"b1"`.
e. Multiply array `"b"` by `100.0` and assign the result to the variable `"b2"`.
f. Print the arrays `"b1"` and `"b2"`.
g. Print `"b1 == b2"`. Are they the same?
h. Why do they display differently? Interrogate the `typecode` of each array to find out why.

2. Let's look at array comparisons.

a. Create an array of values `0` to `9` and assign it to the variable `"arr"`.
b. Print two different ways of expressing the condition where the array is less than 3.
c. Create a numpy condition where `"arr"` is less than 3 OR greater than 8.
d. Use the `"where"` function to create a new array where the value `is "arr * 5"` if the above condition is `True` and `"arr * -5"` where the condition is `False`.

3. Let's implement a mathematical function that works on arrays.

a. Write a function that takes a 2-D array of horizontal zonal (east-west) wind components (*u*, in m/s) and a 2-D array of horizontal meridional (north-south) wind components (*v* , in m/s) and returns an array of the magnitudes of the total wind. Include a test for the overall magnitude: if it is less than 0.1 then set it equal to 0.1. (We might presume this particular domain has no non-zero winds and that only winds above 0.1 m/s constitute "good" data while those below are indistinguishable from the minimum due to noise.)

   The return value should be an array of the same shape and type as the input arrays. The magnitude of the wind can be calculated as the square root of the sum of the squares of the *u* and *v* winds.
b. Test your function on *u* and *v* winds of values: `[[4, 5, 6], [2, 3, 4]]` and `[[2, 2, 2], [1, 1, 1]]`
c. Test your function on *u* and *v* winds of values: `[[4, 5, 0.01], [2, 3, 4]]` and `[[2, 2, 0.03], [1, 1, 1]]`. Does your default minimum magnitude get used?

Solution 3: Array calculations and operations

1.

```
>>> import numpy as np
>>> a = np.array([range(4), range(10, 14)])
>>> b = np.array([2, -1, 1, 0])
>>> print(a * b)
[[  0  -1   2   0]
 [ 20 -11  12   0]]
>>> b1 = b * 100
>>> b2 = b * 100.0
>>> print(b1, b2)
[ 200 -100  100    0] [ 200. -100.  100.    0.]
>>> print(b1 == b2)
[ True  True  True  True]
>>> print(b1.dtype, b2.dtype)
int64 float64
```

2.

```
>>> arr = np.arange(10)
>>> print(arr < 3)
[ True  True  True False False False False False False False]
>>> print(np.less(arr, 3))
[ True  True  True False False False False False False False]
>>> condition = np.logical_or(arr < 3, arr > 8)
>>> new_arr = np.where(condition, arr * 5, arr * -5)
>>> print(new_arr)
[  0   5  10 -15 -20 -25 -30 -35 -40  45]
```

3.

```
>>> def calcMagnitude(u, v, minmag = 0.1):
...     mag = ((u**2) + (v**2))**0.5
...     output = np.where(mag > minmag, mag, minmag)
...     return output

>>> u = np.array([[4, 5, 6], [2, 3, 4]])
>>> v = np.array([[2, 2, 2], [1, 1, 1]])
>>> print(calcMagnitude(u, v))
[[ 4.47213595  5.38516481  6.32455532]
 [ 2.23606798  3.16227766  4.12310563]]

>>> u = np.array([[4, 5, 0.01], [2, 3, 4]])
>>> v = np.array([[2, 2, 0.03], [1, 1, 1]])
>>> print(calcMagnitude(u, v))
[[ 4.47213595  5.38516481  0.1       ]
 [ 2.23606798  3.16227766  4.12310563]]
```

Exercise 4: Working with missing values

Aim: An introduction to masked arrays to represent missing values

Issues covered:

- Creating a masked array
- Masking certain values in an array
- Using the "`masked_where`" function to create a masked array
- Applying a mask to an existing array
- Performing calculations with masked arrays

1. Let's create a masked array and play with it.

a. Import the "`numpy.ma`" module as "`MA`".
b. Create a masked array from a list of values (`0` to `9`) with a `fill_value` of `-999`).
c. Print the array to view its values. Print the "`fill_value`" attribute.
d. Mask the third value in the array using "`MA.masked`".
e. Print the array to view how it has changed.
f. Print the mask associated with the array (i.e. "`marr.mask`").
g. Create a new array called "`narr`" that is equal to "`marr`" where "`marr`" is less than `7` and masked otherwise).
h. Print the array to view its values.
i. Print its missing value `("narr.fill_value").`
j. Print an array that has converts "`narr`" so that the missing values are represented by the missing value. What type of array is this?

2. Let's create a mask that is smaller than the overall array.

a. Create a masked array of values `1` to `8` and assign it to the variable "`m1`". Print the result.
b. Re-shape the array to the shape (2, 4) and assign it to the variable "`m2`". Print the result.
c. Mask values of "`m2`" greater than `6` and assign the result to the variable "`m3`". Print the result.
d. Multiply "`m3`" by `100` and print the result.
e. Subtract a normal numpy array of `ones` that is the same shape (i.e. (2, 4)) from "`m3`". Is the result a normal array or a masked array?

Solution 4: Working with missing values

1.

```
>>> import numpy.ma as MA
>>> marr = MA.masked_array(range(10), fill_value = -999)
>>> print(marr, marr.fill_value)
[0 1 2 3 4 5 6 7 8 9] -999
>>> marr[2] = MA.masked
>>> print(marr)
[0 1 -- 3 4 5 6 7 8 9]
>>> print(marr.mask)
[False False  True False False False False False False False]
>>> narr = MA.masked_where(marr > 6, marr)
>>> print(narr)
[0 1 -- 3 4 5 6 -- -- --]
>>> x = MA.filled(narr)
>>> print(x)
[     0      1 999999      3      4      5      6 999999 999999 999999]
>>> print(type(x))
<type 'numpy.ndarray'>
```

2.

```
>>> m1 = MA.masked_array(range(1, 9))
>>> print(m1)
[1 2 3 4 5 6 7 8]
>>> m2 = m1.reshape(2, 4)
>>> print(m2)
[[1 2 3 4]
 [5 6 7 8]]
>>> m3 = MA.masked_greater(m2, 6)
>>> print(m3)
[[1 2 3 4]
 [5 6 -- --]]
>>> res = m3 - np.ones((2, 4))
>>> print(res)
[[0.0 1.0 2.0 3.0]
 [4.0 5.0 -- --]]
>>> print(type(res))
<class 'numpy.ma.core.MaskedArray'>
```