Arrays

NumPy provides the core data type for numerical analysis – arrays. NumPy arrays are widely used through the Python econsystem and are extended by other key libraries including pandas, an essential library for data analysis.

In [2]:

```
from numpy import array
```

In [3]:

```
x=[0.0,1,2,3,4]
y=array(x)
```

In [4]:

```
y
```

Out[4]:

```
array([0., 1., 2., 3., 4.])
```

In [5]:

```
type(y)
```

Out[5]:

```
numpy.ndarray
```

two (or higher)-dimensional arrays using nested list.

In [6]:

```
y=array([[0.0,1,2,3,4],[5,6,7,8,9]])
y
```

Out[6]:

```
array([[0., 1., 2., 3., 4.],
       [5., 6., 7., 8., 9.]])
```

In [7]:

```
len(y)
```

Out[7]:

```
2
```

In [21]:

```
y = array([[[1,2],[3,4]],[[5,6],[7,8]]])
y
```

Out[21]:

```
array([[[1, 2],
        [3, 4]],

       [[5, 6],
        [7, 8]]])
```

In [22]:

```
len(y)
```

Out[22]:

```
2
```

Array dtypes

Homogeneous arrays can contain a variety of numeric data types. The most common data type is float64 (or double), which corresponds to the python built-in data type of float (and C/C++ double). By default, calls to array will preserve the type of the input, if possible. If an input contains all integers, it will have a dtype of int32 (similar to the built-in data type int). If an input contains integers, floats, or a mix of the two, the array's data type will be float64. If the input contains a mix of integers, floats and complex types, the array will be initialized to hold complex data

In [9]:

```
x=[0,1,2,3,4] #integer
y=array(x)
y.dtype
```

Out[9]:

```
dtype('int32')
```

In [10]:

```
x=[0.0,1,2,3,4] #0.0 is float
y=array(x)
```

In [11]:

```
y.dtype
```

Out[11]:

```
dtype('float64')
```

In [12]:

```
x=[0.0+1j ,1,2,3,4] #(0.0 +1j) is a complex
y=array(x)
```

In [13]:

```
y
```

Out[13]:

```
array([0.+1.j, 1.+0.j, 2.+0.j, 3.+0.j, 4.+0.j])
```

In [14]:

```
y.dtype
```

Out[14]:

```
dtype('complex128')
```

NumPy attempts to find the smallest data type which can represent the data when constructing an array. It is possible to force NumPy to select a particular dtype by using the keyword argument dtype=datatype when initializing the array.

In [15]:

```
x=[0,1,2,3,4] #integer

y=array(x)
```

In [16]:

```
y.dtype
```

Out[16]:

```
dtype('int32')
```

In [17]:

```
y=array(x,dtype="float64") #striing dtype
y.dtype
```

Out[17]:

```
dtype('float64')
```

In [18]:

```
y = array(x, dtype="float32") # NumPy type dtype
y.dtype
```

Out[18]:

```
dtype('float32')
```

1-dimensional Arrays

A vector x = [1 2 3 4 5] is entered as a 1-dimensional array using

In [19]:

```
x=array([1.0,2.0,3.0,4.0,5.0])
x
```

Out[19]:

```
array([1., 2., 3., 4., 5.])
```

If an array with 2-dimensions is required, it is necessary to use a trivial nested list.

In [23]:

```
x=array([[1.0,2.0,3.0,4.0,5.0]])
x
```

Out[23]:

```
array([[1., 2., 3., 4., 5.]])
```

In [25]:

```
len(x)
```

Out[25]:

```
1
```

In [26]:

```
x = array([[1.0],[2.0],[3.0],[4.0],[5.0]])
x
```

Out[26]:

```
array([[1.],
       [2.],
       [3.],
       [4.],
       [5.]])
```

2-dimensional Arrays

Two-dimensional arrays are rows of columns, and so x = □

□

1 2 3 4 5 6 7 8 9 □

□, is input by enter the array one row at a time, each in a list, and then encapsulate the row lists in another list

In [27]:

```
x = array([[1.0,2.0,3.0],[4.0,5.0,6.0],[7.0,8.0,9.0]])
x
```

Out[27]:

```
array([[1., 2., 3.],
       [4., 5., 6.],
       [7., 8., 9.]])
```

Multidimensional Arrays

Higher dimensional arrays have a number of uses, for example when modeling a time-varying covariance. Multidimensional (N-dimensional) arrays are available for N up to about 30, depending on the size of each dimension. Manually initializing higher dimension arrays is tedious and error prone, and so it is better to use functions such as zeros((2, 2, 2)) or empty((2, 2, 2)).

Matrix

Matrices are essentially a subset of arrays and behave in a virtually identical manner. The matrix class is deprecated and so should not be used. While NumPy is likely to support the matrix class for the forseeable future, its use is discouraged. In practice, there is no good reason to not use 2-dimensional arrays. The two important differences are: • Matrices always have 2 dimensions • Matrices follow the rules of linear algebra for ∗ 1- and 2-dimensional arrays can be copied to a matrix by calling matrix on an array. Alternatively, mat or asmatrix provides a faster method to coerce an array to behave like a matrix without copying any data.

In [143]:

```
import numpy as np
x=[0.0,1,2,3,4] #any float makes all float
y=array(x)
type(y)
```

Out[143]:

```
numpy.ndarray
```

In [29]:

```
y*y #elementry-by-element
```

Out[29]:

```
array([ 0.,  1.,  4.,  9., 16.])
```

In [144]:

```
z=np.asmatrix(x)
type(z)
```

Out[144]:

```
numpy.matrix
```

Concatenation

Concatenation is the process by which one array is appended to another. Arrays can be concatenation horizon□tally or vertically. For example, suppose x = □ 1 2 3 4 □

and y = □ 5 6 7 8 □ and z =

□ x y □ needs to be constructed. This can be accomplished by treating x and y as elements of a new array and using the function concatenate to join them. The inputs to concatenate must be grouped in a tuple and the key□word argument axis specifies whether the arrays are to be vertically (axis = 0) or horizontally (axis = 1) concatenated.

In [32]:

```
import numpy as np
x=array([[1.0,2.0],[3.0,4.0]])
y=array([[5.0,6.0],[7.0,8.0]])
z=np.concatenate((x,y),axis=0)
z
```

Out[32]:

```
array([[1., 2.],
       [3., 4.],
       [5., 6.],
       [7., 8.]])
```

In [35]:

```
z=np.concatenate((x,y),axis=1)
z
```

Out[35]:

```
array([[1., 2., 5., 6.],
       [3., 4., 7., 8.]])
```

Concatenating is the code equivalent of block forms in linear algebra. Alternatively, the functions vstack and hstack can be used to vertically or horizontally stack arrays, respectively

In [37]:

```
z=np.vstack((x,y)) #same as z=concatenate((x,y),axis=0)
```

In [38]:

```
z
```

Out[38]:

```
array([[1., 2.],
       [3., 4.],
       [5., 6.],
       [7., 8.]])
```

In [40]:

```
z=np.hstack((x,y)) #same as z=z=concatenate((x,y),axis=1)
z
```

Out[40]:

```
array([[1., 2., 5., 6.],
       [3., 4., 7., 8.]])
```

Scalar Selection

Pure scalar selection is the simplest method to select elements from an array, and is implemented using [i] for 1-dimensional arrays, [i, j] for 2-dimensional arrays and [i1,i2,...,in] for general n-dimensional arrays. Like all indexing in Python, selection is 0-based so that [0] is the first element in a 1-d array, [0,0] is the upper left element in a 2-d array, and so on.

In [41]:

```
x=array([1.0,2.0,3.0,4.0,5.0])
x[0]
```

Out[41]:

```
1.0
```

In [42]:

```
x=array([[1.0,2,3],[4,5,6]])
x
```

Out[42]:

```
array([[1., 2., 3.],
       [4., 5., 6.]])
```

In [43]:

```
x[1,2]
```

Out[43]:

```
6.0
```

In [45]:

```
type(x[1,2])
```

Out[45]:

```
numpy.float64
```

Pure scalar selection always returns a single element which is not an array. The data type of the selected element matches the data type of the array used in the selection. Scalar selection can also be used to assign values in an array.

In [46]:

```
x=array([1.0,2.0,3.0,4.0,5.0])
x[0]=-5
x
```

Out[46]:

```
array([-5.,  2.,  3.,  4.,  5.])
```

Array Slicing

Basic slicing of 1-dimensional arrays is identical to slicing a simple list, and the returned type of all slicing operations matches the array being sliced.

In [50]:

```
x = array([1.0,2.0,3.0,4.0,5.0])
y = x[:]
y
```

Out[50]:

```
array([1., 2., 3., 4., 5.])
```

In [52]:

```
 y = x[:2]
y
```

Out[52]:

```
array([1., 2.])
```

2-dimensional arrays,

In [53]:

```
y = array([[0.0, 1, 2, 3, 4],[5, 6, 7, 8, 9]])
y
```

Out[53]:

```
array([[0., 1., 2., 3., 4.],
       [5., 6., 7., 8., 9.]])
```

In [54]:

```
 y[:1,:]
```

Out[54]:

```
array([[0., 1., 2., 3., 4.]])
```

In [55]:

```
y[1:]
```

Out[55]:

```
array([[5., 6., 7., 8., 9.]])
```

In [57]:

```
y[:1,:1]
```

Out[57]:

```
array([[0.]])
```

Mixed Selection using Scalar and Slice Selectors

In [59]:

```
x = array([[1.0,2],[3,4]])
x[:1,:1]
y
```

Out[59]:

```
array([[0., 1., 2., 3., 4.],
       [5., 6., 7., 8., 9.]])
```

In [60]:

```
x[0,:1]
x
```

Out[60]:

```
array([[1., 2.],
       [3., 4.]])
```

In [61]:

```
x = array([[0.0, 1, 2, 3, 4],[5, 6, 7, 8, 9]])
x[:1,:] # Row 0, all columns, 2-dimensional
```

Out[61]:

```
array([[0., 1., 2., 3., 4.]])
```

In [62]:

```
np.ndim(x[0,:])
```

Out[62]:

```
1
```

In [65]:

```
np.ndim(x[0,0])
```

Out[65]:

```
0
```

Mixed Selection using Scalar and Slice Selectors

When arrays have more than 1-dimension, it is often useful to mix scalar and slice selectors to select an entire row, column or panel of a 3-dimensional array. This is similar to pure slicing with one important caveat – dimensions selected using scalar selectors are eliminated. For example, if x is a 2-dimensional array, then x[0,:] will select the first row. However, unlike the 2-dimensional array constructed using the slice x[:1,:], x[0,:] will be a 1-dimensional array.

In [68]:

```
x=array([[1.0,2],[3,4]])
```

In [69]:

```
x
```

Out[69]:

```
array([[1., 2.],
       [3., 4.]])
```

In [70]:

```
x[:1,:]
```

Out[70]:

```
array([[1., 2.]])
```

In [71]:

```
x[0:,1]
```

Out[71]:

```
array([2., 4.])
```

Assignment using Slicing

Slicing and scalar selection can be used to assign arrays that have the same dimension as the slice

In [72]:

```
x=array([[0.0]*3]*3) #repeats the list 3 times
x
```

Out[72]:

```
array([[0., 0., 0.],
       [0., 0., 0.],
       [0., 0., 0.]])
```

In [75]:

```
x[0,:]=array([1.0,2.0,3.0])
```

In [76]:

```
x
```

Out[76]:

```
array([[1., 2., 3.],
       [0., 0., 0.],
       [0., 0., 0.]])
```

In [77]:

```
 x[::2,::2]
```

Out[77]:

```
array([[1., 3.],
       [0., 0.]])
```

In [78]:

```
 x[::2,::2] = array([[-99.0,-99],[-99,-99]]) # 2 by 2
```

In [79]:

```
x
```

Out[79]:

```
array([[-99.,   2., -99.],
       [  0.,   0.,   0.],
       [-99.,   0., -99.]])
```

In [81]:

```
x[1,1]=np.pi
```

In [82]:

```
x
```

Out[82]:

```
array([[-99.        ,   2.        , -99.        ],
       [  0.        ,   3.14159265,   0.        ],
       [-99.        ,   0.        , -99.        ]])
```

NumPy attempts to automatic (silent) data type conversion if an element with one data type is inserted into an array with a different type. For example, if an array has an integer data type, placing a float into the array results in the float being truncated and stored as an integer. This is dangerous, and so in most cases, arrays should be initialized to contain floats unless a considered decision is taken to use a different data type

In [83]:

```
x = [0, 1, 2, 3, 4] # Integers
```

In [84]:

```
y=array(x)
```

In [85]:

```
y.dtype
```

Out[85]:

```
dtype('int32')
```

In [86]:

```
y[0]=3.141592
y
```

Out[86]:

```
array([3, 1, 2, 3, 4])
```

In [87]:

```
x = [0.0, 1, 2, 3, 4] # 1 Float makes all float
y=array(x)
```

In [88]:

```
y.dtype
```

Out[88]:

```
dtype('float64')
```

In [89]:

```
y[0] = 3.141592
y
```

Out[89]:

```
array([3.141592, 1.      , 2.      , 3.      , 4.      ])
```

Linear Slicing using flat Data in arrays is stored in row-major order – elements are indexed by first counting across rows and then down columns. For example, in the 2-dimensional array x = □ □ 1 2 3 4 5 6 7 8 9 □ □ the first element of x is 1, the second element is 2, the third is 3, the fourth is 4, and so on. In addition to slicing using the [:,:,...,:] syntax, k-dimensional arrays can be linear sliced. Linear slicing assigns an index to

In [93]:

```
y = np.reshape(np.arange(25.0),(5,5))
y
```

Out[93]:

```
array([[ 0.,  1.,  2.,  3.,  4.],
       [ 5.,  6.,  7.,  8.,  9.],
       [10., 11., 12., 13., 14.],
       [15., 16., 17., 18., 19.],
       [20., 21., 22., 23., 24.]])
```

In [94]:

```
y[0] # Same as y[0,:], first row
```

Out[94]:

```
array([0., 1., 2., 3., 4.])
```

In [95]:

```
y.flat[0] # Scalar slice, flat is 1-dimensional
```

Out[95]:

```
0.0
```

In [ ]:

```
y[6] # Error
```

In [96]:

```
y.flat[6] # Element 6
```

Out[96]:

```
6.0
```

In [97]:

```
y.flat[12:15]
```

Out[97]:

```
array([12., 13., 14.])
```

In [98]:

```
y.flat[:] # All element slice
```

Out[98]:

```
array([ 0.,  1.,  2.,  3.,  4.,  5.,  6.,  7.,  8.,  9., 10., 11., 12.,
       13., 14., 15., 16., 17., 18., 19., 20., 21., 22., 23., 24.])
```

Slicing and Memory Management

Unlike lists, slices of arrays do not copy the underlying data. Instead, a slice of an array returns a view of the array which shares the data in the sliced array. This is important since changes in slices will propagate to the original array as well as to any other slices which share the same element.

In [100]:

```
x = np.reshape(np.arange(4.0),(2,2))
x
```

Out[100]:

```
array([[0., 1.],
       [2., 3.]])
```

In [101]:

```
 s1 = x[0,:] # First row
```

In [102]:

```
s2 = x[:,0] # First column
```

In [103]:

```
s1
```

Out[103]:

```
array([0., 1.])
```

In [104]:

```
 s2
```

Out[104]:

```
array([0., 2.])
```

In [105]:

```
x
```

Out[105]:

```
array([[0., 1.],
       [2., 3.]])
```

```
If changes should not propagate to parent and sibling arrays, it is necessary to call
copy on the slice. Alterna•tively, they can also be copied by calling array on an
existing array.
```

In [108]:

```
x = np.reshape(np.arange(4.0),(2,2))
s1[0]=-3.14
s1
```

Out[108]:

```
array([-3.14,  1.  ])
```

In [109]:

```
s2
```

Out[109]:

```
array([-3.14,  2.  ])
```

In [111]:

```
x[0,0]
```

Out[111]:

```
0.0
```

There is one notable exception to this rule – when using pure scalar selection the (scalar) value returned is always a copy.

In [113]:

```
x = np.arange(5.0)
y = x[0] # Pure scalar selection
z = x[:1] # A pure slice
y = -3.14
y # y Changes
```

Out[113]:

```
-3.14
```

In [114]:

```
 x # No propagation
```

Out[114]:

```
array([0., 1., 2., 3., 4.])
```

In [115]:

```
 z # No changes to z either
```

Out[115]:

```
array([0.])
```

In [116]:

```
z[0] = -2.79
y # No propagation since y used pure scalar selection
```

Out[116]:

```
-3.14
```

In [117]:

```
 x # z is a view of x, so changes propagate
```

Out[117]:

```
array([-2.79,  1.  ,  2.  ,  3.  ,  4.  ])
```

Finally, assignments from functions which change values will automatically create a copy of the underlying array.

In [118]:

```
x = array([[0.0, 1.0],[2.0,3.0]])
y=x
```

In [119]:

```
print(id(x),id(y)) # Same id, same objec
```

```
2409417643280 2409417643280
```

In [120]:

```
y = x + 1.0
y
```

Out[120]:

```
array([[1., 2.],
       [3., 4.]])
```

In [122]:

```
print(id(x),id(y)) # Different

```

```
2409417643280 2409417227408
```

In [123]:

```
x
```

Out[123]:

```
array([[0., 1.],
       [2., 3.]])
```

Even trivial function such as y = x + 0.0 create a copy of x, and so the only scenario where explicit copying is required is when y is directly assigned using a slice of x, and changes to y should not propagate to x.

import and Modules

Python, by default, only has access to a small number of built-in types and functions. The vast majority of functions are located in modules, and before a function can be accessed, the module which contains the function must be imported. For example, when using %pylab in an IPython session a large number of modules are automatically imported, including NumPy, SciPy, and matplotlib. While this style of importing

useful for learning and interactive use, care is needed to make sure that the correct module is imported when designing more complex programs. For example, both NumPy and SciPy have functions called sqrt and so it is not clear which will be used by Pylab. import can be used in a variety of ways. The simplest is to use from module import $*$ which imports all functions in module. This method of using import can dangerous since it is possible for functions in one module to be hidden by later imports from other modeuls. A better method is to only import the required functions. This still places functions at the top level of the namespace while preventing conflicts

In [125]:

```
from pylab import log2 # Will import log2 only
from scipy import log10 # Will not import the log2 from SciPy
```

The functions log2 and log10 can both be called in subsequent code. An alternative and more common method is to use import in the form

In [126]:

```
import pylab
import scipy
import numpy
```

which allows functions to be accessed using dot-notation and the module name, for example scipy.log2. It is also possible to rename modules when imported using as

In [127]:

```
import pylab as pl
import scipy as sp
import numpy as np
```

```
The only difference between the two types is that import scipy is implicitly calling
import scipy as scipy.
When this form of import is used, functions are used with the “as” name. For example,
the square root provided
by SciPy is accessed using sp.sqrt, while the pylab square root is pl.sqrt. Using this
form of import allows
both to be used where appropriate.
```

Calling Functions

```
Functions calls have different conventions than most other expressions. The most
important difference is that
functions can take more than one input and return more than one output. The generic
structure of a function
call is out1, out2, out3, ... = functionname(in1, in2, in3, ...). The important aspects
of this structure are
• If multiple outputs are returned, but only one output variable is provided, the output
will (generally) be a
tuple.
• If more than one output variable is given in a function call, the number of output
must match the number
of output provided by the function. It is not possible to ask for two output if a
function returns three –
using an incorrect number of outputs results in ValueError: too many values to unpack.
• Both inputs and outputs must be separated by commas (,)
```

• Inputs can be the result of other functions. For example, the following are equivalent,

In [129]:

```
y=np.var(x)
```

In [130]:

```
np.mean(y)
```

Out[130]:

```
1.25
```

In [132]:

```
np.mean(np.var(x))
```

Out[132]:

```
1.25
```

Required Arguments

Most functions have required arguments. For example, consider the definition of array from help(array)

In [133]:

```
array(object, dtype=None, copy=True, order=None, subok=False, ndmin=0)
```

Out[133]:

```
array(<class 'object'>, dtype=object)
```

In [134]:

```
 array([[1.0,2.0],[3.0,4.0]])
```

Out[134]:

```
array([[1., 2.],
       [3., 4.]])
```

In [135]:

```
 array([[1.0,2.0],[3.0,4.0]], 'int32')
```

Out[135]:

```
array([[1, 2],
       [3, 4]])
```

Keyword Arguments

All of the arguments to array can be called by the keyword that appears in the help file definition.

In [136]:

```
array(object=[[1.0,2.0],[3.0,4.0]])
array([[1.0,2.0],[3.0,4.0]], dtype=None, copy=True, order=None, subok=False)
```

Out[136]:

```
array([[1., 2.],
       [3., 4.]])
```

In [138]:

```
array(dtype='complex64', object = [[1.0,2.0],[3.0,4.0]], copy=True)
```

Out[138]:

```
array([[1.+0.j, 2.+0.j],
       [3.+0.j, 4.+0.j]], dtype=complex64)
```

Default Arguments

Functions have defaults for optional arguments. These are listed in the function definition and appear in the
help in the form keyword=default. Returning to array, all inputs have default arguments except object – the
only required input.

Multiple Outputs

Some functions can have more than 1 output. These functions can be used in a single output mode or in multiple
output mode. For example, shape can be used on an array to determine the size of each dimension.

In [140]:

```
x = array([[1.0,2.0],[3.0,4.0]])
s=np.shape(x)
```

In [141]:

```
s
```

Out[141]:

```
(2, 2)
```

In [142]:

```
u=[1,1,2,3,5,8]
u
```

Out[142]:

```
[1, 1, 2, 3, 5, 8]
```

In [147]:

```
v=np.vstack([1,1,2,3,5,8])
v
```

Out[147]:

```
array([[1],
       [1],
       [2],
       [3],
       [5],
       [8]])
```

In [152]:

```
x=np.vstack([1,0,0,1])
```

In [153]:

```
x
```

Out[153]:

```
array([[1],
       [0],
       [0],
       [1]])
```

In [164]:

```
import numpy as np

u = np.array([1, 1, 2, 3, 5, 8])

v = np.array([
    [1, 1],
    [2, 3],
    [5, 8]
])

x = np.array([
    [1, 0],
    [0, 1]
])

y = np.array([
    [1, 2],
    [3, 4]
])

z = np.array([
    [1, 2, 1, 2],
    [3, 4, 3, 4],
    [1, 2, 1, 2]
])

w = np.block([
    [x, x],
    [y, y]
])

print("u =", u)
print("v =", v)
print("x =", x)
print("y =", y)
print("z =", z)
print("w =", w)
```

```
u = [1 1 2 3 5 8]
v = [[1 1]
 [2 3]
 [5 8]]
x = [[1 0]
 [0 1]]
y = [[1 2]
 [3 4]]
z = [[1 2 1 2]
 [3 4 3 4]
 [1 2 1 2]]
w = [[1 0 1 0]
 [0 1 0 1]
 [1 2 1 2]
 [3 4 3 4]]
```

Q2. . What command would select x from w? (Hint: w[?,?] is the same as x.)

In [165]:

```
x_from_w = w[:2, :2]
```

Q3.. What command would select [x 0 y 0] 0 from w? Is there more than one? If there are, list all alternatives.

In [166]:

```
selected_subarray = w[[0, 2], :]
```

In [167]:

```
selected_subarray = w[::2, :]
```

Q4. . What command would select y from z? List all alternatives.

In [158]:

```
selected_subarray = z[1:3, :]
```

In [159]:

```
selected_subarray = z[1:, :]
```

5. Explore the options for creating an array using keyword arguments. Create an array containing y = □ 1 −2 −3 4 □with combination of keyword arguments in: (a) dtype in float, float64, int32 (32-bit integers), uint32 (32-bit unsigned integers) and complex128 (double precision complex numbers). (b) copy either True or False. (c) ndim either 3 or 4. Use shape(y) to see the effect of this argument.

In [160]:

```
import numpy as np

# float
y_float = np.array([[1, -2], [-3, 4]], dtype=float)

# float64 (double precision float)
y_float64 = np.array([[1, -2], [-3, 4]], dtype=np.float64)

# int32 (32-bit integers)
y_int32 = np.array([[1, -2], [-3, 4]], dtype=np.int32)

# uint32 (32-bit unsigned integers)
y_uint32 = np.array([[1, -2], [-3, 4]], dtype=np.uint32)

# complex128 (double precision complex numbers)
y_complex128 = np.array([[1, -2], [-3, 4]], dtype=np.complex128)
```

In [170]:

```
# With copy=True (default behavior)
y_copy_true = np.array([[1, -2], [-3, 4]], copy=True)
y_copy_true

# With copy=False (shares data with the original)
y_copy_false = np.array([[1, -2], [-3, 4]], copy=False)
```

In [162]:

```
# Create a 2D array (default)
y_2d = np.array([[1, -2], [-3, 4]])

# Create a 3D array using reshape
y_3d = y_2d.reshape((1, 2, 2))

# Create a 4D array using reshape
y_4d = y_2d.reshape((1, 1, 2, 2))

# Check the shapes
print("Shape of y_2d:", y_2d.shape)
print("Shape of y_3d:", y_3d.shape)
print("Shape of y_4d:", y_4d.shape)
```

```
Shape of y_2d: (2, 2)
Shape of y_3d: (1, 2, 2)
Shape of y_4d: (1, 1, 2, 2)
```

Q6.. Enter y = [1.6180 2.7182 3.1415] as an array. Define x = mat(y). How is x different from y?

In [163]:

```
import numpy as np

# Create the array y
y = np.array([1.6180, 2.7182, 3.1415])

# Define x as a matrix (2D array) using mat()
x = np.mat(y)

# Print the values of x and y
print("x:", x)
print("y:", y)
```

```
x: [[1.618  2.7182 3.1415]]
y: [1.618  2.7182 3.1415]
```

In [ ]:

```
```