

SEE1002

Introduction to Computing for Energy and Environment

Part 4: Python for Science and Engineering
Section 2: Introduction to NumPy

Course Outline

Part 1: Introduction to computing

Part 2: Elements of Python programming

Section 1: Fundamentals

Section 2: Branching or Decision Making

Section 3: Loops

Section 4: Functions

Part 3: Basic Python programming

Section 1: Modules

Section 2: Structure of a Python program

Section 3: Good programming practices

Part 4: Python for science and engineering

Section 1: File input and output

Section 2: Vectors, matrices and arrays

Section 3: Other topics

Outline

1. Motivation
2. Basics
3. Standard operations
4. Methods and functions

Overview

Virtually all applications of computing in science and engineering involve the use of vectors and matrices. In this section we're going to explain how they're handled in Python.

I. Background

Review

- Before explaining how they're defined in Python, let's review some basic ideas about vectors and matrices.
- Note: this is not going to be a comprehensive review!

Vectors

- A **vector** has a magnitude and direction. Alternatively it is just a **set of coordinates**:

$$\mathbf{U} = (u, v, w)$$

$$\mathbf{F} = (F_x, F_y, F_z)$$

- Vectors obey a number of standard **vector operations**.

$$\mathbf{u_{tot}} = \mathbf{U}_1 + \mathbf{U}_2 = (u_1 + u_2, v_1 + v_2, w_1 + w_2)$$

$$\text{work} = (F_x, F_y, F_z) \cdot (\Delta x, \Delta y, \Delta z)$$

$$= F_x \Delta x + F_y \Delta y + F_z \Delta z$$

- Addition is trivial but multiplication is more complicated. Vectors can be multiplied using a **scalar or vector product**.

Vector multiplication

- There are various ways of multiplying vectors. In your physics and mathematics classes, you have been introduced to the scalar (dot) and vector (cross) products:

$$(a_1, a_2, a_3) \cdot (b_1, b_2, b_3) = a_1 b_1 + a_2 b_2 + a_3 b_3$$

$$(a_1, a_2, a_3) \times (b_1, b_2, b_3) = (a_2 b_3 - a_3 b_2, a_3 b_1 - a_1 b_3, a_1 b_2 - a_2 b_1)$$

- One can also define an elementwise product:

$$(a_1, a_2, a_3) \circ (b_1, b_2, b_3) = (a_1 b_1, a_2 b_2, a_3 b_3)$$

Matrix

- A **matrix** A or A_{ij} generalises a vector to two dimensions. In physics, it shows how a quantity **varies in space**. It can be used to represent a **field**.

$$A = \begin{pmatrix} A_{1,1} & A_{1,2} & \cdots & A_{1,j} \\ A_{2,1} & A_{2,2} & \cdots & A_{2,j} \\ \vdots & \vdots & \ddots & \vdots \\ A_{i,1} & A_{m,2} & \cdots & A_{i,j} \end{pmatrix}$$

typically the indices refer to **spatial locations** (e.g. $i \rightarrow x$, $j \rightarrow y$).

Matrix operations

- There are many matrix operations which you should have learned about in linear algebra. Here are a few of them:

$$A = A_{ij}, \quad A^T = A_{ji}$$

$$C = AB = \sum_{i=1}^M \sum_{j=1}^N \sum_{k=1}^N \sum_{l=1}^O A_{ij} B_{kl}$$

$$AI = A$$

$$AA^{-1} = I$$

Using lists to represent vectors and matrices in Python

- Using a separate variable for each element isn't efficient!
- The obvious approach would be to use list. More precisely, we would need to use a **list for vectors** and a **list of lists for matrices**.
- This will work but it's not very convenient...

Example 1: Representing vectors using lists

```
u1x=1
u1y=0
u1z=0

u2x=0
u2y=1
u2z=0
u1=[u1x,u1y,u1z]
u2=[u2x,u2y,u2z]

print (u1)
print (u2)
```

```
[1, 0, 0]
[0, 1, 0]
```

Defining vectors using lists is trivial.

Example I continued: vector operations

```
# vector addition
utot = []
for el1,el2 in zip(u1,u2):
    utot.append(el1+el2)
print (utot)
utot2 = [u1x+u2x,u1y+u2y,u1z+u2z]
print (utot2)

# dot product
udot = 0.0
for el1,el2 in zip(u1,u2):
    udot += el1*el2
print (udot)
```

```
[1, 1, 0]
[1, 1, 0]
0.0
```

Using lists it is not possible to add or multiply the vectors directly.

Example 2: Representing a matrix using a list of lists

Define a 3x3 matrix whose elements are given by the integers from 1-9.

```
nrows = 3
ncols = 3
A = []

count = 0
for i in range(nrows):
    rowvector = []
    for j in range(ncols):
        count += 1
        rowvector.append(count)
    A.append(rowvector)

print (A)
```

```
In [50]: A
Out[50]: [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

N.B. this doesn't really look like a matrix!

Defining matrices is more complicated

Other matrix operations

- Defining a matrix already takes a fair bit of work.
- If we want to do everything from scratch, we also need to write functions for the various matrix operations.
 - ▶ Matrix add of A_{ij} and B_{ij} : loop over i and j
 - ▶ Matrix multiplication of A_{jk} and B_{kl} : loop over j, k, l ;
 - ▶ etc.

Disadvantages of using lists

1. We need to define the structure of the vector or array using for loops.
 2. We also need to implement the various operations manually, e.g. vector addition requires a for loop.
 3. In principle we could define functions for the various operations or use somebody else's library.
 4. The properties of vectors and matrices aren't naturally reflected.
- We're going to use a module that provides a data structure for vectors and arrays as well as functions (methods) for the various operations. *This is going to save us a lot of time!*

Desirable features of a module

To save us from having to do a lot of extra work, we would like to use a module for vectors and arrays that has the following features:

1. Vectors or matrices can be defined without using for loops.
2. Easy to access and modify individual elements
3. Functions for the standard operations already exist.
4. The vectors or matrices can be referred to as an object without having to use a for loop (e.g. $A+B$, $f(A)$, etc).

2. Basics

Introduction to NumPy

We can use the NumPy module to work with vectors or matrices. It provides us with:

1. Data structures
2. Function/methods
3. Data abstraction (e.g. refer to A rather than specific elements of A)

NumPy simplifies things a lot!!

Invoking NumPy

As NumPy is a module it can be invoked in just the same way as the other modules that we have used.

```
import numpy
import numpy as np
from numpy import *
```

The second form is the most common. The third form should be used only for short programs. *The reasons for this will become clear!*

Arrays in numpy

In computer science, one doesn't distinguish between vectors and matrices. Both are referred to by the generic term **array**.

Henceforth we will focus on the **NumPy array type**.

i) Creating arrays

There are number of ways in which NumPy arrays can be created.

Creating an array in NumPy - I

- A NumPy array can be created by analogy with lists.
- All we need to do is put `np.array()` around the list.

```
arrayname = np.array(list)
```

- If `list` is a 2-D list of lists then `np.array()` creates a 2-D matrix.

Example 3: creating a list using numpy.array

```
import numpy as np
stdlist=[0,1,2]
nplist=np.array(stdlist)
print ('list version=',stdlist)
print ('numpy array version=',nplist)
```

```
list version= [0, 1, 2]
numpy array version= [0 1 2]
```

```
In [117]: stdlist?
Type:      list
String form: [0, 1, 2]
Length:    3
```

```
In [295]: nplist?
```

```
Type:      ndarray
String form: [0 1 2]
Length:    3
```

list has commas

numpy list doesn't have commas

Array ordering

- For a simple (1-d) list, NumPy will output a list with exactly the same shape.
- For a list of lists, a matrix will be created. But **which index corresponds to the row and which index corresponds to the column?**
 - ▶ By default, the innermost list corresponds to the columns of a matrix.

```
In [35]: print( np.array( [ [0,1,2], [3,4,5], [6,7,8] ] ) )  
[[0 1 2]  
 [3 4 5]  
 [6 7 8]]
```

Digression: storage of arrays in memory

- For obvious reasons, **arrays occupy adjacent locations in memory.**
- In Python and most programming languages it's the last index, i.e. the column, that corresponds to adjacent locations. This is referred to as **row-major order**.
- Equivalently: **the column index varies fastest.**
- This explains why the innermost list corresponds to the column!

Creating an array in NumPy -2

- The previous approach only works for small arrays.
- For real problems we need to create a big array of a specified size.
- A zero array can be created using `numpy.zeros`.

```
arrayname = np.zeros(shape [, type])
```

where *shape* specifies the shape of the array and *type* specifies the data type.

- By default, the output is a NumPy array containing floats.

Example 4: creating zero arrays

vector

```
import numpy as np

rowvect=np.zeros(10)
matrix1=np.zeros( (2,3))
matrix2=np.zeros( (3,3))
print ('rowvect=',rowvect)
print ('matrix1=',matrix1)
print ('matrix2=',matrix2)
```

```
rowvect= [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
matrix1= [[ 0.  0.  0.]
 [ 0.  0.  0.]]
matrix2= [[ 0.  0.]
 [ 0.  0.]
 [ 0.  0.]
```

matrix

The decimal point indicates that these arrays are composed of floats.

Example 5: creating specific array types

```
import numpy as np

rowvect=np.zeros(10)
rowvect_i=np.zeros(10,dtype='int')
rowvect_f=np.zeros(10,dtype='float')

print( 'rowvect=',rowvect )
print( 'rowvect_i=',rowvect_i )
print( 'rowvect_f=',rowvect_f )
```

```
rowvect= [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
rowvect_i= [0 0 0 0 0 0 0 0 0 0]
rowvect_f= [0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
```

int



float



Specifying shape

- To specify a 1-D array with N elements:

```
arrayname = np.zeros(N)
```

- To specify a 2-D MxN array:

```
arrayname = np.zeros((M,N))
```

- N.B. it's conventional to specify the shape using a tuple.
But one can also use a list.

Row vectors versus column vectors

- At heart, numpy arrays treat multidimensional arrays as (1-D) vectors. This is why we can reshape them so easily.
- This is convenient for analysing data but potentially confusing when dealing with linear algebra problems.
- We'll return to this later, but for now we simply note that NumPy arrays don't distinguish between row and column vectors.

Specifying type

By default NumPy creates an array with floats.

- To specify an array of integers:

```
arrayname = np.zeros(shape, dtype='int')
```

- To specify an array of floats:

```
arrayname = np.zeros(shape, dtype='float')
```

- N.B. it's possible to specify different kinds of ints and floats, but we won't cover this here.

Creating an array in NumPy -3

- Instead of specifying the shape of the array through the shape parameters, one can resize it using `reshape`.

```
arrayname = np.zeros(N*M)  
arrayname.reshape(N,M)
```

```
arrayname = np.zeros((N,M))
```

- The approaches are completely equivalent. But the first emphasizes that as far as Python is concerned, all arrays are 1-D. Reshaping arrays is very useful.

Understanding `numpy.reshape`

- `numpy.reshape` can be used to change the shape (number of dimensions) of a numpy array. For a 1-D array with $N \times M$ elements:

```
newarray = arrayname.reshape( (N,M) )
```

- Since Python follows row-major order, consecutive elements go into adjacent columns.

```
In [162]: a
Out[162]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

```
In [40]: print(a.reshape(5,2))
[[0 1]
 [2 3]
 [4 5]
 [6 7]
 [8 9]]
```

- -

Automatically creating a range of values

- We are often interested in creating an array that spans a certain range of values, e.g., 1...100, 10, 20, 40, etc.
- Previously we used `range(start, stop, step)` create a list with elements between specified start and stop values. We need to specify the **spacing**.
- We can do similar things with numpy arrays.

numpy.arange

`numpy.arange` is the numpy version of `range`:

```
arrayname = np.arange([start,] stop [,step])
```

where *start* is the starting value, *stop* is the final value, and *step* specifies the spacings. By default the array consists of integers.

- By default, `start=0` and `step=1`
- By contrast with `range`, the data type of return values is determined by the parameters.
- The endpoint is **excluded**.

numpy.linspace

`numpy.linspace` creates an evenly spaced 1-D `numpy` array over a specified interval:

```
arrayname = np.linspace(start, stop[, number])
```

where *start* is the starting value, *stop* is the final value, and *number* (which is optional) specifies the total number of points. Note that if *number* is not specified, Python assumes *number*=50.

- There is **no need to specify the spacing**. This can be convenient.
- The return values are **floats**. By default the **endpoints are included**.

numpy.linspace vs numpy.arange

`numpy.linspace` creates an evenly spaced 1-D
`numpy` array over a specified interval:

```
arrayname = np.linspace(start, stop[, number])
```

- There is **no need to specify the spacing**. This can be convenient.
- The return values are **floats**. By default the **endpoint is included**.

`numpy.arange` is the `numpy` version of `range`:

```
arrayname = np.arange([start,] stop [,step])
```

where *start* is the starting value, *stop* is the final value, and *step* specifies the spacings. By default the array consists of integers.

Example 6a: Using arange

- Use arange to create (i) a vector and (ii) a matrix containing equally spaced ints from 1-100.

```
import numpy as np

N=100
vector=np.arange(1,N+1,1)
print (vector)
```

```
import numpy as np

N=100
Nsqr=10

matrix=np.arange(1,N+1)
matrix=matrix.reshape(Nsqr,Nsqr)
print (matrix)
```

There are several ways of doing part (ii). The simplest is to reshape the vector.

Example 6b: Using linspace

- Use `linspace` to create (i) a vector and (ii) a matrix containing equally spaced floats from 1-100.

```
import numpy as np

N=100
vector=np.linspace(1,N,N)
print (vector)
```

```
import numpy as np

N=100
Nsqr=10

matrix=np.linspace(1,N,N)
matrix=matrix.reshape(Nsqr,Nsqr)
print (matrix)
```


Creating special arrays

`numpy.zeros` is just one way of creating arrays.

```
onearray = np.ones(shape)  
idarray = np.eye(N)  
ranarray = np.random.uniform(low, high, shape)
```

The first command creates an array filled with ones.

The second command creates an NxN identity matrix.

The third command creates a random matrix with values between *low* and *high*.

ii) Assignment

We can assign values to numpy arrays in the same ways as we've done with other variables, i.e., element by element.

```
array[i] = value  
array[i,j] = value    or    array[i][j] = value
```

We can reference multi-dimensional NumPy arrays in two different ways.

Why can we reference NumPy arrays in two different ways?

1. It's more convenient to reference elements in a way that matches standard mathematical notation.

$$A_{ij} \rightarrow A[i, j]$$

2. Traditionally arrays were built from lists of lists. Thus A_{ij} corresponds to the j th element of the list $A[i]$:

$$A_{ij} \rightarrow A[i][j]$$

Example 7: assigning values to 1-D and 2-D arrays

```
import numpy as np

N=4
A1d=np.zeros(N)

for i in range(N):
    A1d[i] = i*i

print (A1d)
```

```
[0.  1.  4.  9.]
```

```
import numpy as np

N=2
A2d=np.zeros((N,N))

count = 0
for i in range(N):
    for j in range(N):
        A2d[i,j] = count
        count = count +1

print (A2d)
```

```
[[ 0.  1.]
 [ 2.  3.]]
```

Multiple assignment

- Assigning values element by element isn't efficient.
- We can save a lot of time by assigning values to a **range** of values.

```
array[istart:istop] = value  
array[:] = value  
array[istart:] = value  
array[:, index] = value
```

Comments

```
array[istart:istop] = value  
array[istart:istop, jstart:jstop] = value
```

- The interpretation of `istart` and `istop` is identical to that for lists.
- `A :` without `start` and `stop` is useful as a placeholder, i.e., to specify all elements in the given direction.
- If only a single index is given, it will be interpreted as a row index.

Vector assignment

- In practice, the values are assigned at the same time.
- In computer science this is referred to as a **vector operation**.
- This is an example of parallel or multicore processing. But for historical reasons this is referred to as **vector assignment**.

Example 8: vector assignment

```
import numpy as np
```

```
N=4
```

```
A1d=np.zeros(N)
```

```
A1d[0:N] = np.pi  
print( A1d )
```

```
A1d[:] = np.pi+1  
print( A1d )
```

```
A1d = np.pi+2  
print( A1d )
```

equivalent

no need to use math.pi

automatic conversion

```
import numpy as np
```

```
N=2
```

```
A2d=np.zeros( (N,N) )
```

```
A2d[0:N, 0:N] = np.pi  
print( A2d )
```

```
A2d[:,:] = np.pi+1  
print()  
print( A2d )
```

```
A2d[0,:] = 0  
print()  
print(A2d)
```

```
A2d[1] = 1  
print()  
print(A2d)
```

final
row

```
[ 3.14159265  3.14159265  3.14159265  3.14159265]  
[ 4.14159265  4.14159265  4.14159265  4.14159265]  
5.14159265359
```

```
[[ 3.14159265  3.14159265]  
 [ 3.14159265  3.14159265]]  
  
[[ 4.14159265  4.14159265]  
 [ 4.14159265  4.14159265]]  
  
[[ 0.          0.          ]  
 [ 4.14159265  4.14159265]]  
  
[[ 0.  0.]  
 [ 1.  1.]]
```


iii) Initialisation and assignment

Python automatically creates a new variable when an assignment is made.

```
x = 1  
x = y
```

Versions of these operations exist in NumPy. However, additional work is required. Assignment of values or **initialization** differs from that for regular variables.

Initialisation

```
In [166]: x=2
```

```
In [167]: x?
```

```
Type:          int  
String form: 2
```

```
In [170]: xarr=np.zeros(1)
```

```
In [171]: xarr[0]=2
```


```
In [172]: xarr?
```

```
Type:          ndarray  
String form: [ 2.]
```


Regular Python variables don't need to be declared before values are assigned (cf. Flowgorithm). However, **numpy arrays must be created before they can be used.**

Assignment versus aliasing

```
In [25]: y=1
In [26]: x=y
In [27]: y=2
In [28]: print(x,y)
1 2
```



```
In [29]: A=np.zeros(2)
In [30]: A[:]=1
In [31]: B=A
In [32]: print(A,B)
[1. 1.] [1. 1.]
In [33]: B[:]=2
In [34]: print(A,B)
[2. 2.] [2. 2.]
```



In NumPy the assignment **does not create a new array but just a new name**. Hence B is just an **alias** for A in the example above. (More formally, A and B reference the same **memory address**.)

Aliasing

```
In [12]: A=np.zeros(2)
```

```
In [13]: A[:]=1
```

```
In [14]: B=A
```

```
In [16]: print(A,B)
[1. 1.] [1. 1.]
```

```
In [17]: B[:]=2
```

```
In [18]: print(A,B)
[2. 2.] [2. 2.]
```

```
import numpy as np
a=np.arange(4).reshape(2,2)
b=a
b=np.eye(2)
print('a=',a)
print('b=',b)
```

```
a= [[0 1]
     [2 3]]
b= [[1. 0.]
     [0. 1.]]
```

```
import numpy as np
a=np.arange(4).reshape(2,2)
b=a
b=b+np.eye(2)
print('a=',a)
print('b=',b)
```

```
a= [[0 1]
     [2 3]]
b= [[ 1.  1.]
     [ 2.  4.]]
```

Alias is broken when
assignment uses B

```
import numpy as np
a=np.arange(4).reshape(2,2)
b=a
b[1,1] = 10.0
b=b+np.eye(2)
print('a=',a)
print('b=',b)
```

```
a= [[ 0  1]
     [ 2 10]]
b= [[ 0  1]
     [ 2 10]]
```

Alias is maintained
when assignment uses
elements of B

In NumPy the assignment **does not** create a new array but just a new **name**. Hence B is just an **alias** for A in the example above. (More formally, A and B reference the same **memory address**.)

Breaking the alias

```
import numpy as np
a=np.arange(4).reshape(2,2)
b=a
b=b+np.eye(2)
print('a=',a)
print('b=',b)
```

```
a= [[0 1]
     [2 3]]
b= [[ 1.  1.]
     [ 2.  4.] ]
```

The alias is broken by a new assignment to the array variable. This turns it into a regular variable. ***Assignment to elements of the array doesn't work.***

Copying

The correct way to assign a variable to another variable in NumPy is to copy it.

```
arrayCopy = array.copy()  
arrayCopy = np.copy(array)
```

This creates a new variable and fills it with the values from the old variables. The new and old variables are independent.

3. Standard operations

Overview

Some of the standard operations that we learned about earlier can also be applied to NumPy arrays.

i) Addition and subtraction

NumPy arrays can be added and subtracted directly. Recall that this can't be done with lists!

```
Cplus = A + B  
Cminus = A - B
```

Example 10

integer arrays

```
import numpy as np

a= np.zeros(10, dtype='int')
b= np.zeros(10, dtype='int')

a[:] = 1
b[:] = 2

print( 'addition:', a+b )
print( 'subtraction:', a[:]-b[:] )
```

```
addition: [3 3 3 3 3 3 3 3 3 3]
subtraction: [-1 -1 -1 -1 -1 -1 -1 -1 -1 -1]
```

the `[:]` is optional

ii) Multiplication

NumPy arrays can be multiple element-by-element using `*`:

```
Cmult = A*B  
Cdiv  = A/B
```

The arrays must have the same shape.

Example I I

float arrays

```
import numpy as np  
a= np.zeros(10)  
b= np.zeros(10)  
  
a[:] = 1  
b[:] = 2  
  
print( 'multiplication:', a*b )  
print( 'division:', a[:]/b[:] )
```

```
multiplication: [ 2.  2.  2.  2.  2.  2.  2.  2.  2.  2.]  
division: [ 0.5  0.5  0.5  0.5  0.5  0.5  0.5  0.5  0.5  0.5]
```

elementwise operations

iii) Other operations

Array elements can be raised to arbitrary powers:

```
Cpow = A**2
```

Arrays can also be compared with binary operators:

```
test = A < B
```

this yields a NumPy array of the same shape as A and B.
Other operators, e.g. `==`, `%`, etc. can be used as well.

Example 12: lookup table revisited

```
import numpy as np

a= np.linspace(1,100,100)
asqrt = a**0.5

for a_el, asqrt_el in zip(a,asqrt):
    print (a_el,asqrt_el)
```

```
93.0 9.64365076099
94.0 9.69535971483
95.0 9.74679434481
96.0 9.79795897113
97.0 9.8488578018
98.0 9.89949493661
99.0 9.94987437107
100.0 10.0
```

This is much easier than using lists!

Example 13: logical tests with binary arrays

```
import numpy as np

a= np.linspace(-20,20,10)
b= np.linspace(1,10,10)

print (a<b)
testflag = a<b

for i,x in enumerate(testflag):
    if x:
        print (b[i],end=' ')
    else:
        print (a[i], end=' ')
```

```
[ True  True  True  True  True  True  True  True False False False]
1.0 2.0 3.0 4.0 5.0 6.0 7.0 11.1111111111 15.5555555556 20.0
```

Occasionally it's useful to work directly with the output of a binary operator.

4. Methods and functions

Overview

Most of the operations on NumPy arrays are associated with a function or **method**. Usually they can be invoked in two different ways:

- **method:** `array.dosomething()`
- **function:** `np.dosomething(array)`

We'll cover both approaches.

We didn't emphasise this previously, but `a.append()` is a method associated with the list `a`.

i) Basic statistics

```
np.mean(array)  
np.min(array)  
np.max(array)  
np.std(array)
```

float



They can also be invoked using methods:

```
array.mean()  
array.min()  
array.max()  
array.std()
```

Example 14

```
import numpy as np

a= np.linspace(1,100,100)
print('a=',a)

print('min=',a.min())
print('max=',a.max())
print('mean=',a.mean())
print('std deviation=',a.std())
```

```
a= [  1.   2.   3.   4.   5.   6.   7.   8.   9.  10.  11.  12.
 13.  14.  15.  16.  17.  18.  19.  20.  21.  22.  23.  24.
 25.  26.  27.  28.  29.  30.  31.  32.  33.  34.  35.  36.
 37.  38.  39.  40.  41.  42.  43.  44.  45.  46.  47.  48.
 49.  50.  51.  52.  53.  54.  55.  56.  57.  58.  59.  60.
 61.  62.  63.  64.  65.  66.  67.  68.  69.  70.  71.  72.
 73.  74.  75.  76.  77.  78.  79.  80.  81.  82.  83.  84.
 85.  86.  87.  88.  89.  90.  91.  92.  93.  94.  95.  96.
 97.  98.  99. 100.]
min= 1.0
max= 100.0
mean= 50.5
std deviation= 28.8660700477
```

Comments

1. Using methods, e.g. `array.mean()`, is preferred Python style.

2. The statistics can be calculated for specific ranges, e.g.

```
a=np.linspace(1,100,100)
```

```
print(np.mean(a))
```

```
a[0:10]
```

```
array([ 1.,  2.,  3.,  4.,  5.,  6.,  7.,  8.,  9., 10.])
```

```
a[0:10].mean()
```

```
5.5
```

ii) Array functions

NumPy versions exist for most of the standard mathematical functions. They return a NumPy array of the same size:

```
np.sin(array)  
np.cos(array)  
np.tan(array)  
np.sqrt(array)  
np.exp(array)  
np.log(array)
```

Note that equivalent numpy methods do not exist for these functions.

Example 15

It's important to note that NumPy functions act differently from regular math functions. This is one of the main reasons why qualified references are a good idea.

```
import numpy as np
import math as math
a= np.linspace(-math.pi,np.pi,5)
print ('a=',a)

print ('np.cos(a)=',np.cos(a))
print ('math.cos(a)=',math.cos(a))
```

```
a= [-3.14159265 -1.57079633  0.          1.57079633  3.14159265]
np.cos(a)= [ -1.00000000e+00  6.12323400e-17  1.00000000e+00  6.12323400e-17
 -1.00000000e+00]
math.cos(a)=
```

TypeError

Traceback (most recent call last)

```
5
6 print 'np.cos(a)=',np.cos(a)
----> 7 print 'math.cos(a)=',math.cos(a)
8 #
9 #for x in a:
```

TypeError: only length-1 arrays can be converted to Python scalars

iii) Array operations - I

There are many, many array operations. Here are the most important unary ones:

```
array.T
```

```
np.linalg.inv()  
np.linalg.det()
```

Note that equivalent numpy methods (e.g. `array.inv()`) do not exist for the functions in the lower box.

Example 16

```
import numpy as np

a= np.linspace(1,9,9).reshape(3,3)
print ('a=',a)
print ('a^T=',a.T)

b = 2.0*np.eye(3)
print
print ('b=',b)
print ('det(b)=',np.linalg.det(b))
print ('inv(b)=',np.linalg.inv(b))
```

```
a= [[ 1.  2.  3.]
     [ 4.  5.  6.]
     [ 7.  8.  9.]]
a^T= [[ 1.  4.  7.]
      [ 2.  5.  8.]
      [ 3.  6.  9.]]
```

```
b= [[ 2.  0.  0.]
     [ 0.  2.  0.]
     [ 0.  0.  2.]]
det(b)= 8.0
inv(b)= [[ 0.5  0.  0. ]
         [ 0.  0.5  0. ]
         [ 0.  0.  0.5]]
```


Column vectors vs. row vectors revisited

```
import numpy as np
a=np.array( [1,0,0] )
print( 'a', a)
print( 'a.T=', a.T)
```

```
a= [1 0 0]
a.T= [1 0 0]
```

```
import numpy as np
b=np.array( [[0,1,2]] )
print( 'b=', b)
print( 'b.T=', b.T)
```

```
b= [[0 1 2]]
b.T= [[0]
      [1]
      [2]]
```

1-D NumPy arrays can't distinguish between column and row vectors. This is because they're just a sequence of numbers. To get around this problem, we need to use 2-D arrays.

iv) Array operations - 2

Here are most important binary array operations:

```
np.dot(a,b)  
np.matmul(a,b)  
np.cross(a,b)
```

dot products (vector)

Matrix multiply

cross products (vector)

Note: `numpy.array` doesn't distinguish between row and column vectors. So `np.dot` will sometimes give you a row vector instead of a column vector.

Example 17

```
import numpy as np

a= np.array([1,0,1])
b= np.array([-1,1,1])
c = np.eye(3)
d = -np.eye(3)

print( 'a.b=', np.dot(a,b) )
print( 'axb=', np.cross(a,b) )
print( 'ac=', np.dot(c,d) )
print( 'ca^T=', np.dot(c,a.T) )
```

```
a.b= 0
axb= [-1 -2  1]
ac= [[-1.  0.  0.]
      [ 0. -1.  0.]
      [ 0.  0. -1.]]
```

v) `numpy.matrix`

You might wonder about our treatment of matrices. It would be nice if we didn't have to bother with numpy functions.

We can do this using `numpy.matrix`. It greatly simplifies linear algebra.

$$C = A * B$$

Furthermore, `numpy.matrix` distinguishes properly between row and column vectors. (With `numpy.array` everything comes out as a row vector.)

Basic operations with `numpy.matrix`

`C = A*B`

Matrix multiply

`C = A+B`

Matrix addition

`A.T`

Transpose

`A.I`

Inverse

Example 18

```
import numpy as np
a = np.matrix( [[0,1], [2,3] ])

print( 'a=', a )
print()

print( 'a^T=', a.T )
print()

print( 'a^{-1}=', a.I )
print()

print( 'a*a^{-1}=', a*a.I )
print()

print( 'a*a^{-1}-I=', a*a.I-np.eye(2) )
```

```
a= [[0 1]
     [2 3]]

a^T= [[0 2]
      [1 3]]

a^{-1}= [[-1.5  0.5]
         [ 1.   0. ]]

a*a^{-1}= [[ 1.  0.]
           [ 0.  1.]]

a*a^{-1}-I= [[ 0.  0.]
             [ 0.  0.]
```

Row vectors versus column vectors using `numpy.matrix`

`numpy.matrix` distinguishes properly between row and column vectors. (With `numpy.array` everything comes out as a row vector.)

```
a = np.matrix( [0,1,2] )
print('a=',a)
print()

print('a^T=',a.T)
print()
```

```
b = np.array( [0,1,2] )
print('b=',b)
print()

print('b^T=',b.T)
print()
```

```
a= [[0 1 2]]
```

```
a^T= [[0]
      [1]
      [2]]
```

```
b= [0 1 2]
```

```
b^T= [0 1 2]
```

vi) File operations

In the previous section we covered how to read and write to files. The procedure was conceptually simple but a bit complicated in practice.

NumPy includes functions for reading and writing to binary and text files. We'll only cover the latter.

a) Reading text files:

A CSV file can be read into a numpy array using `numpy.loadtxt`:

```
array = np.loadtxt('input.txt')
```

By default it's assumed that the columns are separated by spaces and that the data are floats. It's possible to use other delimiters and data types but we won't cover this.

b) Writing text files

A numpy array can be written to a CSV file:

```
np.savetxt('output.txt', array)
```

By default it's assumed that the columns are separated by spaces. It's possible to use other delimiters but we won't cover this.

A CSV file can be read into a numpy array using `numpy.loadtxt`.

Example 19: reading from a text file

93
90
80
77
68

students.dat

```
import numpy as np  
  
array=np.loadtxt('students.dat')  
print( array )
```

```
[93. 90. 80. 77. 68.]
```

Example 20: writing to a text file

93
90
80
77
68

students.dat

9.30000000000000000488e-01
9.00000000000000000222e-01
8.00000000000000000444e-01
7.70000000000000000178e-01
6.80000000000000000488e-01

pct.dat

```
import numpy as np  
  
array=np.loadtxt('students.dat')  
  
pct=array/100  
np.savetxt('pct.dat',pct)
```

Summary

1. The NumPy module should be used for vectors and matrices.
2. It includes data structures (e.g. the numpy array type) and functions.
3. numpy arrays can be created with `np.array()` or `np.zeros`.
4. numpy arrays support vector assignment and functions.
5. `np.copy()` needs to be used to assign the values of one array to another.
6. There are many functions and methods for statistics and linear algebra.