MODULE N5: INTRODUCTION TO NUMPY I

INTRODUCTION TO COMPUTATIONAL PHYSICS

Kai-Feng Chen
National Taiwan University

HERE COMES THE NUMPY

For Python Language

- **NumPy** is the fundamental package for scientific computing with Python. It contains among other things:
  - a powerful N-dimensional array object
  - sophisticated (broadcasting) functions
  - tools for integrating C/C++ and Fortran code
  - useful linear algebra, Fourier transform, and random number capabilities
- Besides its obvious scientific uses, NumPy can also be used as an efficient multi-dimensional container of generic data.

In short: NumPy provides you a convenient **array object + tools**

NUMPY ARRAYS

- In python, the classical idea of arrays has been replaced by a more powerful type: **list**. But sometimes you still need to operate on arrays with a higher efficiency for scientific computing.
- NumPy is acting as an extension to Python for **multi-dimensional arrays**, for example:

```
>>> import numpy as np
>>> a = np.array([1,2,3,4])
>>> a
array([1, 2, 3, 4])
>>> type(a)
<class 'numpy.ndarray'>
```

Such an array can be used to store any potential data from your experimental/theoretical work!

ARRAY CREATION

- Manual construction of 1-dimensional arrays is very simple. For example, you can create an array from a regular Python list or tuple using the `array()` function.

```
>>> a = np.array([1,2,3,4])
>>> a
array([1, 2, 3, 4])
>>> a.ndim
1 ← 1D
>>> a.shape
(4,)
>>> len(a)
4
>>> a.dtype      ← The type of the resulting array is deduced from the
dtype('int64')     type of the elements in the sequences.
>>> a.nbytes    ← Total bytes consumed by the elements
32
```

ARRAY CREATION (II)

- Arrays with higher dimensions (e.g. 2D, 3D, etc.) can be converted from sequences of sequences, or sequences of sequences of sequences, and so on:

```
>>> a = np.array([[1,2,3],[4,5,6]])
>>> a
array([[1, 2, 3],
       [4, 5, 6]])
>>> a.ndim
2 ← 2D
>>> a.shape
(2,3)
>>> len(a) ← Return the size of the 1st dimension
2
>>> b = np.array([[[1,2],[3,4]],[[5,6],[7,8]]])
>>> b.ndim
3 ← 3D
```

ARRAY CREATING FUNCTIONS

- In practice it is not very convenient to enter the numbers one-by-one; NumPy provides several functions to create arrays with some specific contents, e.g. **arange()**, **linspace()**, and **geospace()**:

```
>>> a = np.arange(10)  ← numpy version of range() function
>>> a
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> b = np.linspace(0.,1.,6)  ← starting, ending, # of points
>>> b
array([ 0. ,  0.2,  0.4,  0.6,  0.8,  1. ])
>>> c = np.linspace(0.,1.,5,endpoint=False)
>>> c
array([ 0. ,  0.2,  0.4,  0.6,  0.8])
>>> d = np.geomspace(1.,1000.,4)
>>> d
array([   1.,   10.,  100., 1000.])
```

Unlike the nominal python indexing, the ending point is included by default.

↑ log scale

ARRAY CREATING FUNCTIONS (II)

- Sometimes you just want an array of **zeros**, **ones**, or **identity**, **full**:

```
>>> np.zeros((3,3)) ← note the argument here is a tuple, (3,3)
array([[ 0.,  0.,  0.],
       [ 0.,  0.,  0.],
       [ 0.,  0.,  0.]])
>>> np.ones((3,3))
array([[ 1.,  1.,  1.],
       [ 1.,  1.,  1.],
       [ 1.,  1.,  1.]])
>>> np.eye(3) ← eye = I = identity
array([[ 1.,  0.,  0.],
       [ 0.,  1.,  0.],
       [ 0.,  0.,  1.]])
>>> np.full((3,3), np.inf) ← create an array with filled elements
array([[inf, inf, inf],
       [inf, inf, inf],
       [inf, inf, inf]])
```

ARRAY CREATING FUNCTIONS (III)

- Or, creating an array based on the shape of an existing array with **zeros_like** (ones_like, full_like), with **a defined function** or even **randomly**:

```
>>> a = np.array([1,2,3,4])
>>> np.zeros_like(a)
array([0, 0, 0, 0])
>>>
>>> def f(i,j): return i+j
...
>>> np.fromfunction(f,(3,3))
array([[ 0.,  1.,  2.],
       [ 1.,  2.,  3.],
       [ 2.,  3.,  4.]])
>>>
>>> np.random.rand(3,3) ← note: it's NOT a tuple here.
array([[ 0.19836756,  0.53617863,  0.79492192],
       [ 0.6160475 ,  0.59142948,  0.89777024],
       [ 0.11665536,  0.10973303,  0.04245277]])
```

↑ Random numbers between 0 and 1.

ARRAY DATA TYPES

- You may have noticed that in some of the cases, the array elements could be either integer or float. For example:

```
>>> a = np.array([1,2,3])
>>> a.dtype
dtype('int64')
>>>
>>> b = np.array([1.,2.,3.])
>>> b.dtype
dtype('float64')
>>>
>>> c = np.array([1.,2,3]) ← mix float & integer ⇒ float
>>> c.dtype
dtype('float64')
>>>
>>> d = np.zeros((2,2)) ← Default type is 'float64'.
>>> d.dtype
dtype('float64')
```

Now you see the difference between NumPy array and regular python list! The NumPy array should have a **uniform data type**.

ARRAY DATA TYPES (II)

✤ At the creation of array, you may explicitly specify which data-type you want to use, for example:

```
>>> a = np.array([1,2,3],dtype='int32')
>>> a.dtype
dtype('int32')
>>> a
array([ 1,  2,  3])
>>> b = np.array([1,2,3],dtype='complex128')
>>> b.dtype
dtype('complex128')
>>> b
array([ 1.+0.j,  2.+0.j,  3.+0.j])
>>> c = a+b
>>> c.dtype
dtype('complex128')
```

← will "upgrade" the type if needed

```
>>> c
array([2.+0.j, 4.+0.j, 6.+0.j])
```

Other data types are also available, for example: 'bool', , 'int64', etc.

BASIC OPERATIONS

- In NumPy, arithmetic operations on arrays are **"element-wise"**, ie. <u>one element by one element</u>:

```
>>> a = np.array([1,2,3])
>>> b = np.array([4,5,6])
>>> a**3
array([ 1,  8, 27])
>>> a-b
array([-3, -3, -3])
>>> c = np.array([0,np.pi*0.5,np.pi,np.pi*1.5,np.pi*2])
>>> d = np.sin(c) ← NumPy has all the basic functions you need!
>>> d
array([  0.00000000e+00,   1.00000000e+00,   1.22464680e-16,
        -1.00000000e+00,  -2.44929360e-16])
>>> d>0.5
array([False,  True, False, False, False], dtype=bool)
```

BASIC OPERATIONS (II)

- The * operator is applying on the elements one-by-one as well:

```
>>> a = np.arange(4).reshape(2,2) ← reshape() helps you to
>>> a                                rearrange the shape of array!
array([[0, 1],
       [2, 3]])
>>> b = np.array([[1,1],[2,2]])
>>> a*b
array([[0, 1],
       [4, 6]])
>>> a.dot(b) ← If you want to do matrix product, call dot() function
array([[2, 2],
       [8, 8]])
>>> b.dot(a) ← Non-commute: dot(a,b) ≠ dot(b,a)
array([[2, 4],
       [4, 8]])
```

INDEXING AND SLICING

- The indexing of NumPy array is very close to standard python list:

```
>>> a = np.arange(10)
>>> a
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> a[0], a[1], a[-1]
(0, 1, 9)
>>> b = np.arange(9).reshape((3,3))
>>> b[2]
array([6, 7, 8])
>>> b[2,2]← In 2D, the first dimension corresponds to rows,
8            the second to columns.
>>> b[2,2] = 100
>>> b[2]
array([  6,   7, 100])
```

INDEXING AND SLICING (II)

- Just like nominal python list, it can also be sliced:

```
>>> a = np.arange(10)
>>> a[2:9]
array([2, 3, 4, 5, 6, 7, 8])
>>> a[2:9:2]
array([2, 4, 6, 8])
>>> a[5:] = 0 ← combining slicing & assignment
>>> a
array([0, 1, 2, 3, 4, 0, 0, 0, 0, 0])
>>> b = np.arange(5)
>>> a[5:] = b ← assigned with another array
>>> a
array([0, 1, 2, 3, 4, 0, 1, 2, 3, 4])
>>> a[::2] = 99
>>> a
array([99,  1, 99,  3, 99,  0, 99,  2, 99,  4])
```

Full syntax:
array[start:end:step]

FANCY INDEXING

- NumPy arrays can be indexed with slices, but also with **boolean** or **integer** arrays (*so-called "masks"*). This method is called **fancy indexing**. For example:

```
>>> a = np.arange(10)
>>> a % 2 == 0
array([ True, False,  True, False,  True, False,
True, False,  True, False], dtype=bool)
>>> a[a % 2 == 0]
array([0, 2, 4, 6, 8])
>>> list(range(0,10,2))
[0, 2, 4, 6, 8]
>>> a[range(0,10,2)] = 99
>>> a
array([99,  1, 99,  3, 99,  5, 99,  7, 99,  9])
```

REVIEW: OBJECTS AND VALUES

- If we execute these assignments and following statements:

```
>>> a = 'banana'
>>> b = 'banana'
```

a → 'banana'
b → 'banana'

or

a → 'banana'
b ↗

Same content or same object?

To check whether two variables (a,b) refer to the **SAME** object, one can use the **is** operator (while the regular == operator check the contents).

```
>>> a is b     ← the same object
True
>>> a == b     ← the same content
True
```

Python creates only one 'banana' string in this example.

REVIEW: OBJECTS AND VALUES (II)

- But when you create two lists, you actually get two objects:

```
>>> a = [1,2,3]
>>> b = [1,2,3]
```

```
a ⟶ [1,2,3]
b ⟶ [1,2,3]
```

- In this case we would say that the two lists are equivalent, but not identical, because they are not the same object.

- **"a == b" does not imply "a is b"**:

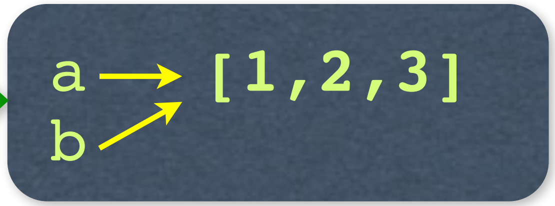
```
>>> a is b
False
>>> a == b
True
```

Python can create two separate lists with the same elements.

REVIEW: ALIASING

- If **a** refers to an object and you assign **b = a**, then both variables refer to the same object:

```
>>> a = [1,2,3]
>>> b = a
>>> a is b
True
```



- The association of a variable with an object is called a **reference**.

- If the aliased object is mutable (such as list!), changes made with one alias affect the other:

```
>>> b[0] = 17
>>> print(a)
[17, 2, 3]
```

Be careful about this when you are developing your code!

VIEW AND COPY

- Remember there are some differences between **alias**, **shallow copy** and **deep copy**. Similar issue happens with NumPy array:

```
>>> a = np.arange(10)
>>> b = a ← b is an alias of a
>>> b is a
True
>>> c = a[2:8] ← c is a view/shallow copy of a
>>> c is a
False
>>> np.may_share_memory(a,c)
True
>>> d = a.copy() ← d is a deep copy of a (call the copy() function)
>>> d is a
False
>>> np.may_share_memory(a,d)
False
```

`may_share_memory()` could tell if two arrays are sharing the same block of memory.

VIEW AND COPY (II)

- Remark: the behavior of slicing is different between nominal Python list and NumPy array!

```
>>> a = np.arange(10)
>>> b = a[2:8]
>>> b[0] = 99
>>> a
array([ 0,  1, 99,  3,  4,  5,  6,  7,  8,  9])
```

NumPy array:
slicing will create a ***view***

```
>>> x = [0,1,2,3,4,5,6,7,8,9]
>>> y = x[2:8]
>>> y[0] = 99
>>> x
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Python list:
slicing will make a copy
(unless the elements are also sequence!)

VIEW AND COPY (III)

✤ However, the operation of fancy indexing will always create a **copy** rather than a **view**:

```
>>> a = np.arange(10)
>>> b = a[2:8]
>>> b[0] = 99
>>> a
array([ 0,  1, 99,  3,  4,  5,  6,  7,  8,  9])
```

Standard slicing will create a ***view***

```
>>> a = np.arange(10)
>>> b = a[range(2,8)]
>>> b[0] = 99
>>> a
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

Fancy indexing will result a ***copy***

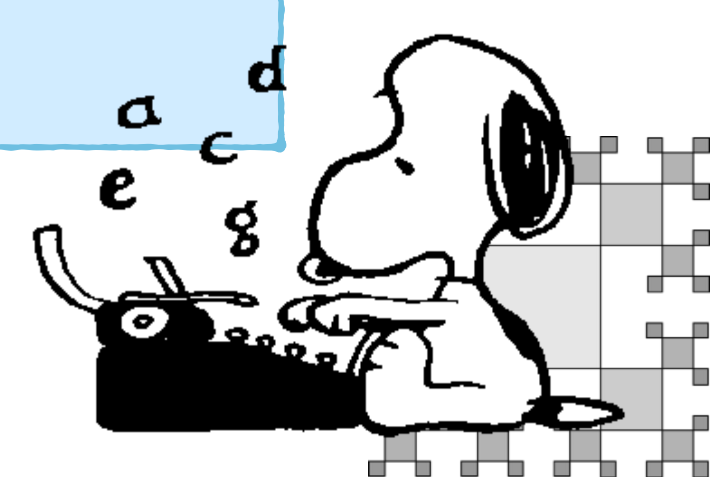
JUST TRY IT OUT!

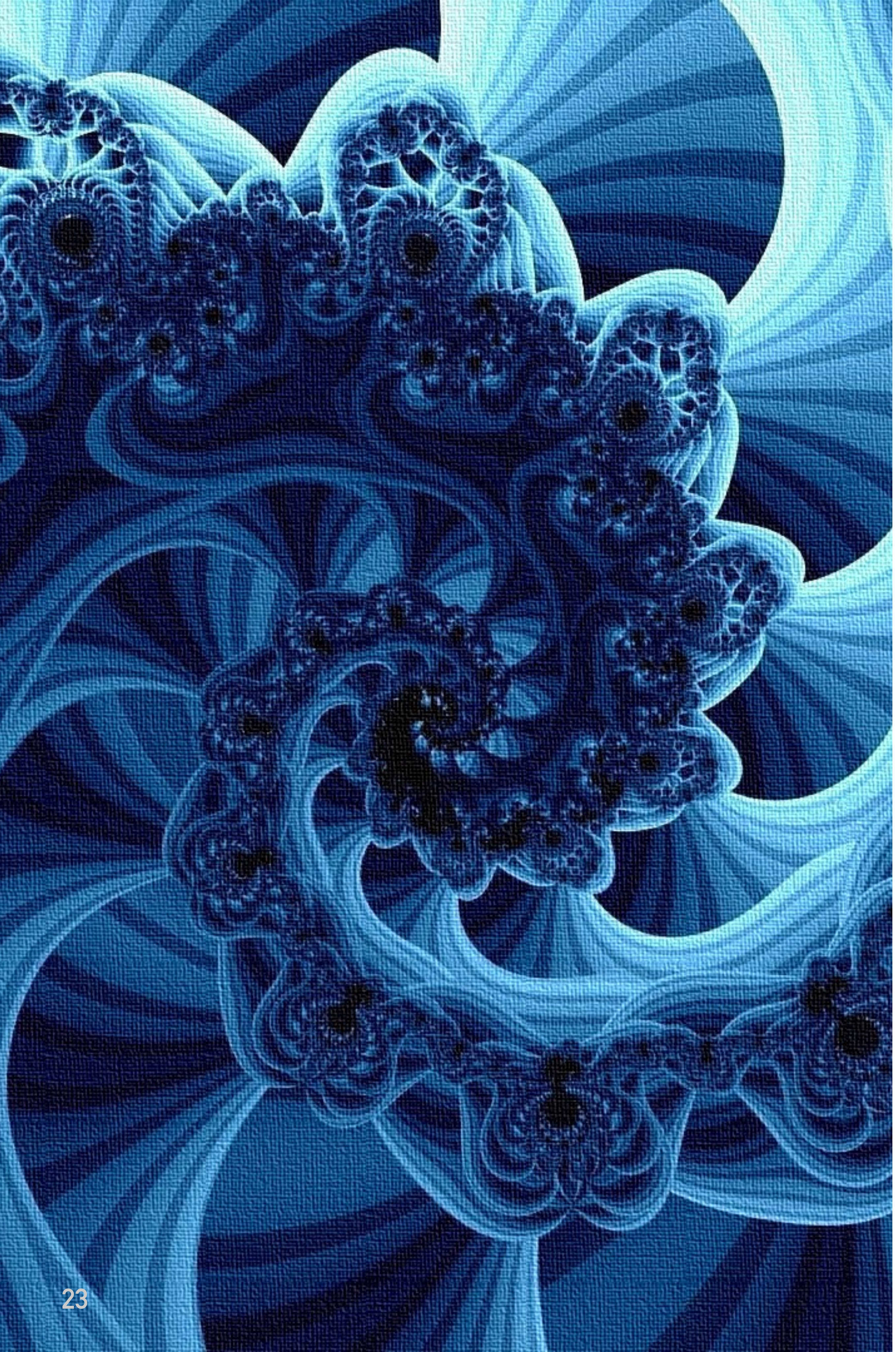
- Remember the full syntax of slicing is **array[start:end:step]**. What will you get if you do this?

```
>>> a = np.arange(10)
>>> a[::-1]
```

- There is another special indexing ellipses operator "…" (very human readable!). See what do you get from a[…,0] and a[0,…]?

```
>>> a = np.arange(81).reshape(9,9)
>>> a[...,0]
>>> a[0,...]
```





MODULE SUMMARY

- In this module we have introduced the concept and basic usage of NumPy arrays, as an extremely useful package for scientific computing with Python language.
- In the next module we will continue to discuss more NumPy array operations.