Lesson N1 – NumPy Basics

Rob Capra
INLS 570

Numerical Python (NumPy)

- Package for scientific computing and data analysis
- Foundation for other tools
- Provides:
  - `ndarray` – fast, space-efficient multidimensional array
  - Fast operation application of math functions to arrays
  - Tools for reading/writing arrays to disk

```
import numpy as np
```

Creating ndarrays

- N-dimensionary array $\begin{bmatrix} 3 & 8 & 2 \\ 7 & 4 & 1 \end{bmatrix}$

- ndarrays can be created using the array() function
- Can create an array from any sequence-like object

```
In [1]: import numpy as np

In [2]: t = [1, 2, 3]

In [3]: a1 = np.array(t)

In [4]: a1
Out[4]: array([1, 2, 3])
```

ndarrays

- ndim, shape, dtype

```
In [10]: t2 = [[1,2,3], [4,5,6]]

In [11]: a2 = np.array(t2)

In [12]: a2
Out[12]:
array([[1, 2, 3],
       [4, 5, 6]])

In [13]: a2.ndim
Out[13]: 2

In [14]: a2.shape
Out[14]: (2, 3)

In [15]: a2.dtype
Out[15]: dtype('int32')
```

Exercise N1.1 – Creating ndarrays

- Use numpy to create the following arrays:

1. Create this array and assign it to a variable ``a1``:

$$\begin{bmatrix} 5 & 9 & 3 \end{bmatrix}$$

2. Create this array and assign it to a variable ``a2``: $\begin{bmatrix} 4 \\ 2 \\ 8 \end{bmatrix}$

3. Create this array and assign it to a variable ``a3``:

$$\begin{bmatrix} 8 & 3 \\ 5 & 2 \\ 4 & 7 \end{bmatrix}$$

4. Write code to print the 9 in ``a1``.
5. Write code to print the 2 in ``a3``.

zeros, ones, empty

- Create arrays of 0's or 1's with a given shape
- Empty does not initialize the values (garbage to start)

```
In [17]: np.zeros((3,6))
Out[17]:
array([[ 0.,  0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.,  0.]])

In [18]: np.ones((2,3))
Out[18]:
array([[ 1.,  1.,  1.],
       [ 1.,  1.,  1.]])

In [19]: np.empty((2,3))
Out[19]:
array([[  2.05915396e+184,   1.77296837e+160,   5.58290476e-091],
       [  4.31091749e-033,   1.00000038e+000,   1.00000000e+000]])
```

zeros_like, ones_like, arange

- Create a new array of the same shape with 0's or 1's
- Arange is like range, but for arrays

```
In [20]: np.arange(7)
Out[20]: array([0, 1, 2, 3, 4, 5, 6])

In [21]: a3 = np.ones((2,3))

In [22]: a3
Out[22]:
array([[ 1.,  1.,  1.],
       [ 1.,  1.,  1.]])

In [23]: a4 = np.zeros_like(a3)

In [24]: a4
Out[24]:
array([[ 0.,  0.,  0.],
       [ 0.,  0.,  0.]])
```

There is also: empty_like

eye, identity

- Create an NxN identity matrix
- 1's on diagonal, 0's elsewhere

```
In [25]: a5 = np.identity(5)
In [26]: a5
Out[26]:
array([[ 1.,  0.,  0.,  0.,  0.],
       [ 0.,  1.,  0.,  0.,  0.],
       [ 0.,  0.,  1.,  0.,  0.],
       [ 0.,  0.,  0.,  1.,  0.],
       [ 0.,  0.,  0.,  0.,  1.]])
In [27]: a5.dtype
Out[27]: dtype('float64')
```

dtype

- Can specify the data type for arrays

```
In [32]: a7 = np.array([1,2,3])
In [34]: a7
Out[34]: array([1, 2, 3])

In [35]: a7.dtype
Out[35]: dtype('int32')

In [36]: a8 = np.array([1,2,3],
dtype=np.float64)

In [37]: a8
Out[37]: array([ 1.,  2.,  3.])

In [38]: a8.dtype
Out[38]: dtype('float64')
```

dtype

- Dtypes are very important
- Mostly, they map to underlying machine data types
- This is a key part of the speed and power of ndarrays
- Because they use underlying machine data types, they can quickly be processed, written as binary data, and integrated with other languages like C.

dtype

Table 4-2. NumPy data types

Type	Type Code	Description
int8, uint8	i1, u1	Signed and unsigned 8-bit (1 byte) integer types
int16, uint16	i2, u2	Signed and unsigned 16-bit integer types
int32, uint32	i4, u4	Signed and unsigned 32-bit integer types
int64, uint64	i8, u8	Signed and unsigned 32-bit integer types
float16	f2	Half-precision floating point
float32	f4 or f	Standard single-precision floating point. Compatible with C float
float64	f8 or d	Standard double-precision floating point. Compatible with C double and Python float object
float128	f16 or g	Extended-precision floating point
complex64, complex128, complex256	c8, c16, c32	Complex numbers represented by two 32, 64, or 128 floats, respectively
bool	?	Boolean type storing True and False values
object	O	Python object type
string_	S	Fixed-length string type (1 byte per character). For example, to create a string dtype with length 10, use 'S10'.
unicode_	U	Fixed-length unicode type (number of bytes platform specific). Same specification semantics as string_ (e.g. 'U10').

Cast using astype

- Convert (cast) array from one type to another

```
In [41]: a9 = np.array([1.2, 2.5, 3.7])
In [42]: a9
Out[42]: array([ 1.2,  2.5,  3.7])

In [43]: a9.dtype
Out[43]: dtype('float64')

In [44]: a10 = a9.astype(np.int32)

In [45]: a10
Out[45]: array([1, 2, 3])

In [46]: a10.dtype
Out[46]: dtype('int32')

In [47]: a11 = a10.astype(np.float64)

In [48]: a11
Out[48]: array([ 1.,  2.,  3.])

In [49]: a11.dtype
Out[49]: dtype('float64')
```

Strings to numbers

- Can also convert strings to numbers this way

```
In [50]: a12 = np.array(['1.2', '2.5', '3.7'],
dtype=np.string_)

In [51]: a12
Out[51]:
array(['1.2', '2.5', '3.7'],
      dtype='|S3')

In [52]: a12.dtype
Out[52]: dtype('S3')

In [53]: a13 = a12.astype(float)

In [54]: a13
Out[54]: array([ 1.2,  2.5,  3.7])

In [55]: a13.dtype
Out[55]: dtype('float64')
```

astype always creates new array (copy of the data), even if the data type is the same as the old type

Arrays versus lists

- Arrays have built-in support for many common operations without having to use for loops

Lists

```
In [56]: t1 = [1,2,3]

In [57]: t2 = []

In [58]: for i in t1:
    ...:     t2.append(i+1)
    ...:

In [59]: t2
Out[59]: [2, 3, 4]

In [60]: t3 = [ i+1 for i in t1 ]

In [61]: t3
Out[61]: [2, 3, 4]
```

NumPy Arrays

```
In [65]: a1 = np.array([1,2,3])

In [66]: a1
Out[66]: array([1, 2, 3])

In [67]: a2 = a1 + 1

In [68]: a2
Out[68]: array([2, 3, 4])
```

More operations

- Arrays and scalars
- Vector operations
- Operations between equal-sized arrays (elementwise)

```
In [69]: a1 = np.array([1,2,3])
In [70]: a1
Out[70]: array([1, 2, 3])

In [71]: a2 = a1 * a1
In [72]: a2
Out[72]: array([1, 4, 9])

In [73]: a3 = a1 * 2
In [74]: a3
Out[74]: array([2, 4, 6])

In [75]: a4 = a1 ** 2
In [76]: a4
Out[76]: array([1, 4, 9])
```

Exercise N1.2 – Operations with ndarrays

- Use numpy to create the following arrays:

1. Create this array and assign it to a variable ``x``: $\begin{bmatrix} 8 & 3 \\ 5 & 2 \\ 4 & 7 \end{bmatrix}$

2. Create this array and assign it to a variable ``y``: $\begin{bmatrix} 1 & 4 \\ 9 & 7 \\ 2 & 3 \end{bmatrix}$

3. Write code to add ``x`` and ``y`` together and print the resulting array.

4. Write code multiply each element of ``x`` by the corresponding element of ``y`` and print the resulting array.

Indexing and Slicing

- Indexing and slices on 1-dim arrays work like lists

```
In [77]: a1 = np.arange(7)

In [78]: a1
Out[78]: array([0, 1, 2, 3, 4, 5, 6])

In [79]: a1[2]
Out[79]: 2

In [80]: a1[3:5]
Out[80]: array([3, 4])

In [81]: a1[:3]
Out[81]: array([0, 1, 2])
```

Broadcasting

- Values can be propagated (or broadcast) into an array

```
In [82]: a1 = np.arange(10)

In [83]: a1
Out[83]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

In [84]: a1[3:5] = 99

In [85]: a1
Out[85]: array([ 0,  1,  2, 99, 99,  5,  6,  7,  8,  9])

In [86]: a1[:3] = 44

In [87]: a1
Out[87]: array([44, 44, 44, 99, 99,  5,  6,  7,  8,  9])
```

Array slices are *views*

- **A BIG difference between array slices and list slices is that array slices are views into the original array.**
- The slice data is not copied – any modifications to the view will be reflected in the original array

```
In [90]: a1 = np.arange(10)
In [91]: a1
Out[91]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

In [92]: fred = a1[3:7]
In [93]: fred
Out[93]: array([3, 4, 5, 6])

In [94]: fred[:3] = 99
In [95]: fred
Out[95]: array([99, 99, 99,  6])

In [96]: a1
Out[96]: array([ 0,  1,  2, 99, 99, 99,  6,  7,  8,  9])
```

Motivation for this is that NumPy is designed to work on large data sets. Copying data for slices would add lots of overhead.

Copy a slice

- If you want to copy a slice, you can copy it explicitly

```
In [97]: a1 = np.arange(10)

In [98]: a1
Out[98]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

In [99]: fred = a1[3:7].copy()

In [100]: fred
Out[100]: array([3, 4, 5, 6])

In [101]: fred[:3] = 99

In [102]: fred
Out[102]: array([99, 99, 99,  6])

In [103]: a1
Out[103]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

Higher dimension indexing

- Things get more complex with higher dimensional arrays

```
In [105]: a1 = np.array([[1,2,3], [4,5,6], [7,8,9]])

In [106]: a1
Out[106]:
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])

In [107]: a1[1]
Out[107]: array([4, 5, 6])

In [108]: a1[1][1]
Out[108]: 5

In [109]: a1[1,1]
Out[109]: 5
```

Higher dimension indexing

```
In [111]: a1 = np.array([[[1,2,3], [4,5,6]], [[7,8,9], [10,11,12]]])
In [112]: a1
Out[112]:
array([[[ 1,  2,  3],
        [ 4,  5,  6]],

       [[ 7,  8,  9],
        [10, 11, 12]]])

In [113]: a1[0]
Out[113]:
array([[1, 2, 3],
       [4, 5, 6]])

In [114]: tmp = a1[0].copy()
```

```
In [115]: a1[0] = 99

In [116]: a1
Out[116]:
array([[[99, 99, 99],
        [99, 99, 99]],

       [[ 7,  8,  9],
        [10, 11, 12]]])

In [117]: a1[0] = tmp

In [118]: a1
Out[118]:
array([[[ 1,  2,  3],
        [ 4,  5,  6]],

       [[ 7,  8,  9],
        [10, 11, 12]]])
```

Higher dimension slicing

- Things get more complex with higher dimensional arrays

```
In [122]: a1 = np.array([[1,2,3],[4,5,6],[7,8,9]])

In [123]: a1
Out[123]:
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])

In [124]: a1[:2]
Out[124]:
array([[1, 2, 3],
       [4, 5, 6]])
```

```
In [125]: a1[:2, 1:]
Out[125]:
array([[2, 3],
       [5, 6]])

In [126]: a1[1, :2]
Out[126]: array([4, 5])

In [127]: a1[2, :1]
Out[127]: array([7])

In [128]: a1[:, :1]
Out[128]:
array([[1],
       [4],
       [7]])
```

Higher dimension indexing

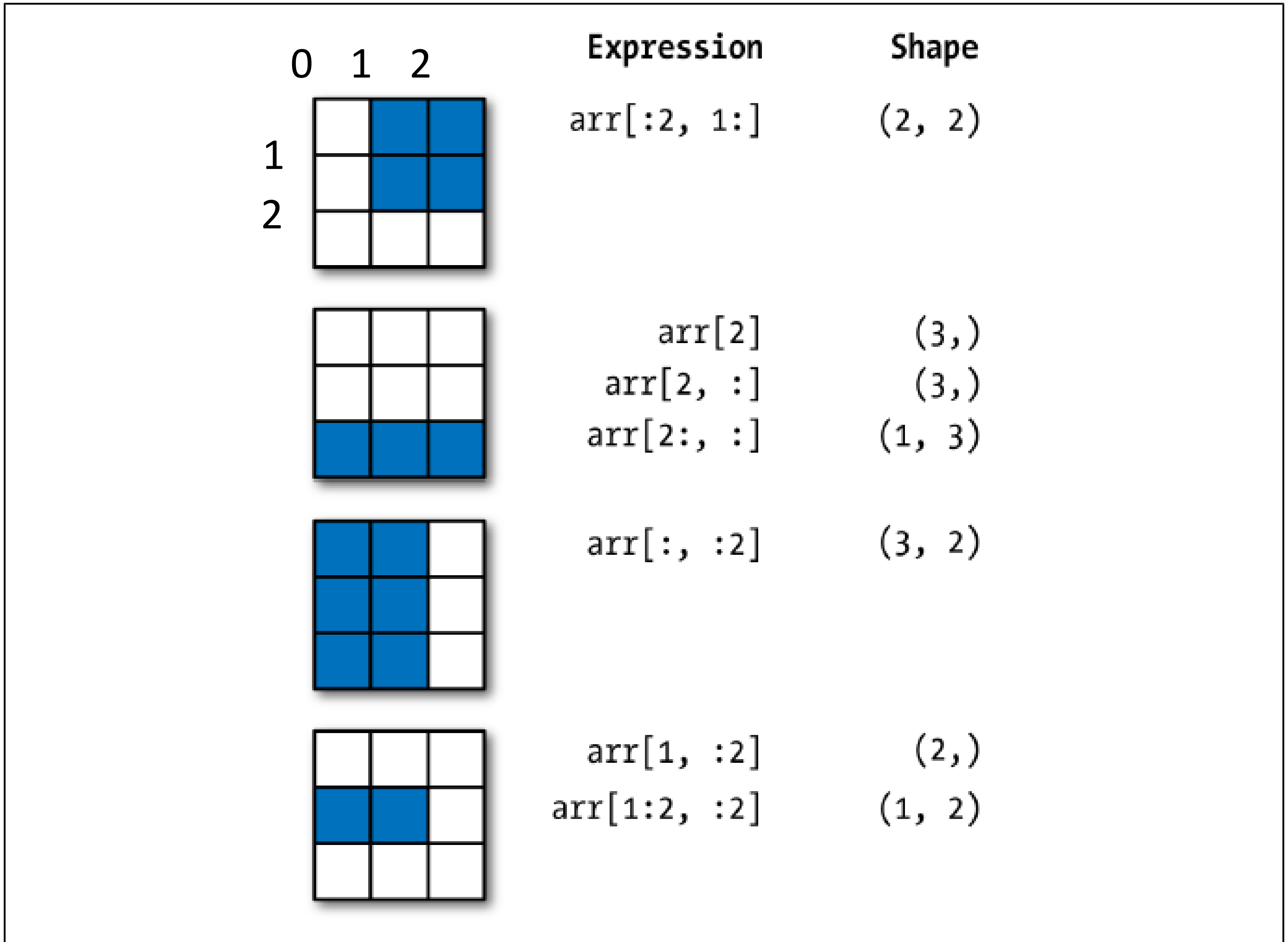

Figure 4-2. Two-dimensional array slicing

Exercise N1.3 – NumPy Arrays

- Add two arrays together and then take a slice of the result

$$\begin{bmatrix} 1 & 2 & 3 & 4 \\ 2 & 3 & 4 & 1 \\ 3 & 4 & 1 & 2 \\ 4 & 1 & 2 & 3 \end{bmatrix} + \begin{bmatrix} 1 & 1 & 2 & 2 \\ 3 & 3 & 4 & 4 \\ 5 & 5 & 6 & 6 \\ 7 & 7 & 8 & 8 \end{bmatrix} = \begin{bmatrix} ? & ? & ? & ? \\ ? & 6 & 8 & ? \\ ? & 9 & 7 & ? \\ ? & ? & ? & ? \end{bmatrix} \xrightarrow[slice]{} \begin{bmatrix} 6 & 8 \\ 9 & 7 \end{bmatrix}$$

a b c

1. Create the first np.array (a)
2. Create the second np.array (b)
3. Add them together and store the result in a new variable (c)
4. Take a slice of c