ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

Lesson 9

ILAI (M1) @ LAAI I.C. @ LM AI

22 October 2024

Michael Lodi

Department of Computer Science and Engineering

These slides draw very heavily from Simone Martini's slides.

Who to contact

For questions on exercises, on Python, tutoring/mentoring

1. *contact* **federico.ruggeri6@unibo.it**

   or

   mohammadrez.hossein3@unibo.it

2. *if you still have questions (eg. Theoretical), contact me: michael.lodi@unibo.it*

ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

Early exam - 8th November, 15:00 - Lab 4 + …

1. a @studio.unibo.it account - mandatory
2. an account on the Ingegneria cluster: https://remo.ing.unibo.it/app/student/infoy (you need the step 1 account to generate the ing account)
3. **At the latest 1 week before** access (with the step 1 account) the website https://eol.unibo.it/

Register for the exam on the Unibo app or Almaesami - **PREFERRED**

- https://almaesami.unibo.it/almaesami/welcome.htm

Only if you are not able to register on app/Almaesami

https://forms.office.com/e/RaYAWQMBP2 (using your @studio account only)

If you have already filled the form but become able to register on AlmaEsami, do so.

Between 7 October and 4 November (included). Late requests will not be accepted.

Essential that you don't skip other lectures of other courses to study for this

Introduction to NumPy

Python and Numpy versions on Virtuale and EOL (on 21/10/24)

Out of my control…

Python 3.10.12 [GCC 11.4.0]

Numpy 1.26.4

In the labs (Thonny), maybe different versions

Our exercises use basic functionalities

NumPy

true n-dimensional arrays

operations on those arrays: math, logical, sorting, etc

- vectorization

discrete Fourier transform

basic linear algebra

basic statistics

basic simulation

Arrays (ndarray):

class numpy.ndarray

Always assume:

```
import numpy as np
```

hence: `np.ndarray`

Fixed size:
if size changes: copying and deletion of older array

Homogeneous: all elements of the same type

but that type could be “Python object” (through indirection)
it is not the intended use

Arrays (`ndarray`)

class ndarray , aliased as array

NOT the class array.array of the standard lib,
which is one-dimensional

We use the function array to build an ndarray

a=np.array([1,2,3]) **argument is a SEQUENCE**

WRONG: np.array(1,2,3)

OK: np.array((1,2,3))

Universal functions (ufunc):

operate in a uniform way on ndarrays

usually element-wise

supporting

array broadcasting, type casting, other "standard" features

A ufunc is a “vectorized” wrapper for a function

Instances of the numpy.ufunc class

Two main mechanisms:

vectorization:
loops are implicit

broadcasting:
in certain situations the shape of an array is “adapted” to the context

Example:

vectorization:

```
a=np.array([1,2,3])
b=np.array([4,5,6])
c=a+b
```

Instead of

```
c=[]
for i in len(a):
    c.append(a[i]+b[i])
```

if a and b are multidimensional:

implicit nested loops

Example:

vectorization:

```
a=np.array([1,2,3])
b=np.array([4,5,6])
c=a+b
```

broadcasting:

```
d=np.array([10])
e=d+a          # e=array([11,12,13])
f=10+a         # f=array([11,12,13])
```

A dimension (an *axis*) is stretched to match the other operand… with some rules

Examples

file lez-1-1.py

Arrays (`ndarray`)

class ndarray , aliased as array

NOT the class array.array of the standard lib,
which is one-dimensional

a=np.array([1,2,3]) **argument is a SEQUENCE**

WRONG: np.array(1,2,3)

OK: np.array((1,2,3))

b=np.array([[1,2,3],[4,5,6]]) **2-dim array**

Arrays (`ndarray`)

class `ndarray`

dimensions

type

are fixed for the entire life of the array

purpose: map a `ndarray` on an array of the underlying (`C`) machine

Arrays: dtypes

May declare the **type** at creation time:

c=np.array([2,3,4], dtype=complex)

dtypes: control on the representation

For the moment let's assume we have int, float, complex, bool

They are *not* Python types: mapped on C type representation

int is not unbounded: mapped into 32 (or 64) bits two's complement integers

May declare the **type** at creation time:

c=np.array([2,3,4], dtype=complex)

[2+0.j, 3+0.j, 4+0.j])

c=np.array([2,3,4], dtype='int8')

Arrays: shape

shape of an array (standard PL terminolgy):

number of dimensions

size of each dimension

C=np.array([[1,2,3],[4,5,6]))

dimensions: 2

shape: (2,3) its a tuple

C.shape

C.ndim x.ndim always equal to len(x.shape)

C.size the number of elements: prod(x.shape)

[prod available in Python 3.8]

C.dtype the dtype of each scalar object

Creating arrays

`np.zeros((3,2))` the arg must be a **shape**

return a zero matrix, `dtype='float64'`

`np.ones((3,2))` return an all-one matrix, `dtype='float64'`

`np.empty((3,2))` return an uninitialized matrix, `dtype= 'float64'`

Optional named parameter `dtype=`…

Creating arrays

NumPy arrays from standard sequences “array like”:

```
x = np.array([2,3,1,0])
x = np.array([[1,2.0],[0,0],(1+1j,3.)])
                          # note mix of tuple and lists, and types
```

```
>>> x
array([[1.+0.j, 2.+0.j],
       [0.+0.j, 0.+0.j],
       [1.+1.j, 3.+0.j]])
>>> x.dtype
dtype('complex128')
>>> type(x)
<class 'numpy.ndarray'>
```

Creating arrays

NumPy arrays from standard sequences “array like”

But careful: if the shape is not correct we will get “strange” results.

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

In fact recent NumPy versions will signal this as

```
ValueError
```

Trick: array([1,2,3,4],[5,6]), dtype=object)

```
>>> array([list([1, 2, 3, 4]), list([5, 6])], dtype=object)
```

This is not the intended use of ndarrays

Creating arrays: `arange`

```
np.arange(first,last,step)
```

like range, but returns

- a 1-dim array, dtype=int64
- last and step are optional
- Same as `np.array(range(…))`

```
np.arange(10,30,5)
        array([10,15,20,25])
```

arange

arange accepts float arguments, even a float step

The dtype of the created array object depends (more or less in the obvious way) from the types of the arguments of arange

In the case of a float step, the size of the created object is non obvious, since it depends on approximations:
(pi on slides is just a shorthand for the actual float (884279719003555 / 281474976710656))

```
from numpy import pi
np.arange(pi, 3*pi, pi)
        array([pi, 2pi])
np.arange(pi,3*pi+10**-15, pi)
        array([pi, 2pi, 3pi])
np.arange(pi,3*pi+10**-16, pi)
        array([pi, 2pi])
```

linspace

linspace

generates an uniformly spaced array:

np.linspace(0,2,9) 9 numbers from 0 to 2 (included)

array([0., 0.25, 0.5 , 0.75, 1., 1.25, 1.5 , 1.75, 2.])

Arrays from generators: `fromiter`

An efficient way to create an array is to use a generator

```
>>> G=(i**2 for i in range(10))
>>> a=np.fromiter(G,dtype=int)
>>> a
array([ 0,  1,  4,  9, 16, 25, 36, 49, 64, 81])
```

Arrays from functions: `fromfunction`

```
def f(i,j):
    return i == j
def g(i,j):
    return i+j
```

```
ba=np.fromfunction(f,(3,3))
>>> ba
array([[ True, False, False],
       [False,  True, False],
       [False, False,  True]])
```

```
ca=np.fromfunction(g,(3,3))
>>> ca
array([[0., 1., 2.],
       [1., 2., 3.],
       [2., 3., 4.]])
```

warning: not from every function. See the slide at the end…

`reshape` arrays

Method

`A.reshape(`*`shape`*`)`

gives *`shape`* to the array `A`

```
np.arange(10,30,5).reshape((2,2))
        array([[10, 15],
               [20, 25]])
```

shape must match the size of A!

`reshape` gives a *view* on the *base array*

Arrays: indexing

ndarrays: indexed (elements accessed)
like standard Python sequences

n-dimensional ndarrays: the n indexes may appear in the same […] (more efficient)

Arrays: indexing

```
>>> x=np.array([[1,2,3,4],[5,6,7,8]])
array([[1, 2, 3, 4],
       [5, 6, 7, 8]])
>>> x[0][2]
3
>>> x[0,2]          #more efficient;
                    #x[0][2] first construct x[0]
3
>>> x[0]
array([1, 2, 3, 4])
>>> x[0]=np.array([5,5,5,5])
>>> x
array([[5, 5, 5, 5],
       [5, 6, 7, 8]])
```

Arrays: slices

Use slices like in standard Python

```
>>> x = np.arange(10)
>>> x[2:5]
array([2, 3, 4])
>>> x[:-7]
array([0, 1, 2])
>>> x[1:7:2]
array([1, 3, 5])
```

Arrays: slices

```
>>> y = np.arange(35).reshape(5,7)
>>> y
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, 25, 26, 27],
       [28, 29, 30, 31, 32, 33, 34]])
>>> y[1:5:2,::3]
array([[ 7, 10, 13],
       [21, 24, 27]])
>>> y[:,1]       # the second (ie, index 1) column of y
array([ 1,  8, 15, 22, 29])
```

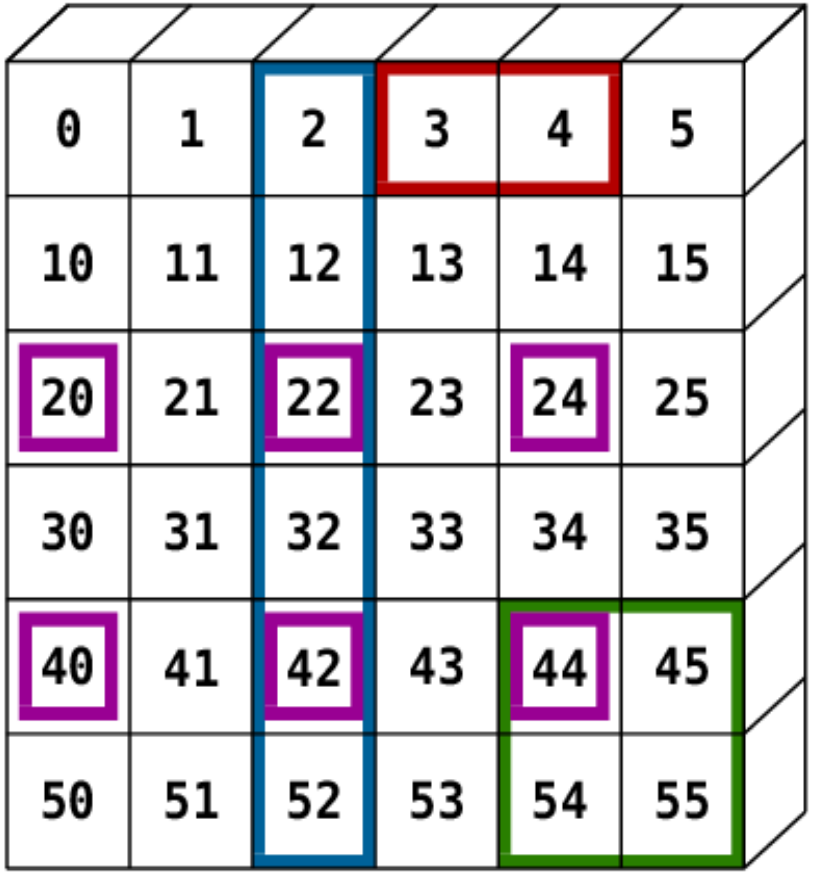
Arrays: slices, example

```
>>> a[0, 3:5]
array([3, 4])

>>> a[4:, 4:]
array([[44, 55],
       [54, 55]])

>>> a[:, 2]
a([2, 12, 22, 32, 42, 52])

>>> a[2::2, ::2]
array([[20, 22, 24],
       [40, 42, 44]])
```



https://scipy-lectures.org/_images/numpy_indexing.png

break until 10.30

Arrays: slices

Use like slices in Python

But:

may use slice LHS of =, but **can never make the array grow/shrink**

hence size of RHS must match size of LHS slice (or… se later)

indexing of entire dimensions, and a fortiori slices

produces views of the original, base array

Arrays: slices

indexing of entire dimensions, and a fortiori slices

produces views of the original, base array

think to explicit copy(), if needed

On standard lists

```
L=[1,2,3,4,5]
LL=L[1:2]
LL[0]=100
```

L is not changed

On ndarrays:

```
X=np.array([1,2,3,4,5])
XX=X[1:2]
XX[0]=100
```

X is changed

now X is [1,100,3,4,5]

slices produce *views*

```
>>> x=np.array([[1,2,3,4],[5,6,7,8]])
array([[1, 2, 3, 4],
       [5, 6, 7, 8]])
>>> y = x[0]
>>> x[0,1]=99
>>> y
array([ 1, 99,  3,  4])
>>> y.base
array([[ 1, 99,  3,  4],
       [ 5,  6,  7,  8]])
>>> c1=x[:,-1]
>>> c1
array([4, 8])
>>> x[-1,-1]=88
>>> c1
array([ 4, 88])
>>> c1.base
array([[ 1, 99,  3,  4],
       [ 5,  6,  7, 88]])
```

views

slices

entire dim selection

reshape

***produces views on the* base array**

hence lots of aliasing is around!

Fancy indexing

Fancy = Indirect

use arrays as index

integer arrays

Boolean arrays (Boolean filtering)

Warning:

fancy indexing produces *copies !*

(unless used as LHS in an assignment)

Fancy indexing: index arrays

Indirect indexing via arrays

Arrays *of integer type* may be used as indexes

the *contents* of the index array are the real indexes

```
>>> x = np.arange(10,1,-1)
>>> x
array([10,  9,  8,  7,  6,  5,  4,  3,  2])
>>> x[np.array([3, 3, 1, 8])]
array([7, 7, 9, 2])  #not a view, a new array
>>> inda=np.array([3, 3, 1, 8])
>>> x[inda]
array([7, 7, 9, 2])  #not a view, a new array
```

Fancy indexing: index arrays

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

Negative values: OK as usual

```
>>> x[np.array([3,3,-3,8])]
array([7, 7, 4, 2])
```

Index out of bounds: ERROR (contrary to slices)

```
>>> x[np.array([3, 3, 20, 8])]
<type 'exceptions.IndexError'>:
index 20 out of bounds 0<=index<9
```

Fancy indexing: index arrays

Multidimensional index array

Returned:

same shape as the index array

of course, type and values of the array being indexed

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

>>> x[np.array([[1,1],[2,3]])]
array([[9, 9],
       [8, 7]])
```

Fancy indexing: index arrays

Indexing multidimensional arrays

especially when also the index array is multidimensional

see the docs (*too complex to think in N-Dimensions*)

Fancy indexing: Boolean arrays

If indexing array is a “Boolean array”:

Booleans “select” the elements in the indexed array

True: select

False: do no select

Very handy tool!

Fancy indexing: Boolean arrays

```
>>> np.random.seed(3)
>>> a = np.random.randint(0, 21, 15)
        #caveat: 21 NOT INCLUDED ☹
>>> a
array([10,  3,  8,  0, 19, 10, 11,  9, 10,  6,
0, 20, 12,  7, 14])
>>> (a % 3 == 0)      #vectorization!
array([False,  True, False,  True, False, False,
False,  True, False, True,  True, False,  True,
False, False])
>>> a[a % 3 == 0]
array([ 3,  0,  9,  6,  0, 12])
```

Fancy indexing: LHS of =

Warning:

fancy indexing produces *copies !*

unless used as LHS in an assignment

```
>>>a=np.arange(10)
>>>a[a%3==0]=99
```

Fancy indexing: LHS of =

Warning:

fancy indexing produces *copies !*

unless used as LHS in an assignment

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

Fancy indexing: LHS of =

Warning:

fancy indexing produces *copies !*

unless used as LHS in an assignment

```
>>>a=np.arange(10)
>>>a[a%3==0]=99
>>>a
array([99,  1,  2, 99,  4,  5, 99,  7,  8, 99])
>>>a[np.array([1,2])]=100
>>>a
array([99, 100, 100, 99, 4, 5, 99, 7, 8, 99])
```

Boolean filtering : example

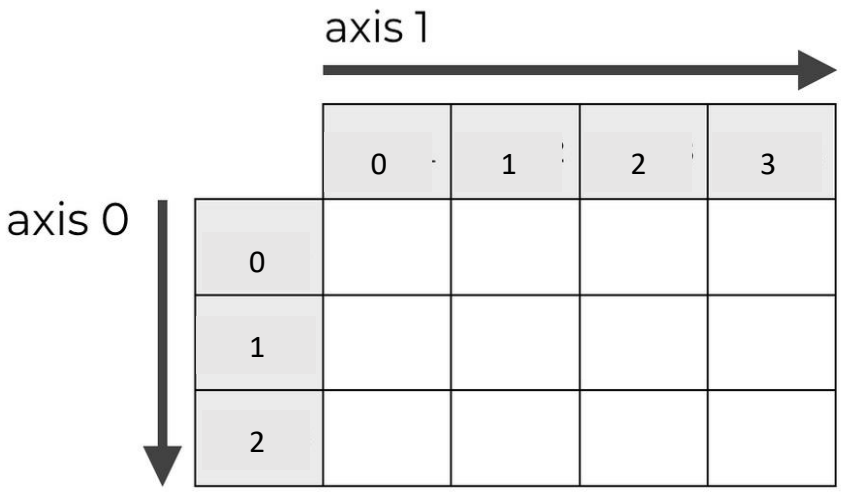
Add a constant to all negative elements:

```
>>> x = np.array([1., -1., -2., 3])
>>> x[x < 0] += 20
>>> x
array([  1.,  19.,  18.,   3.])
```

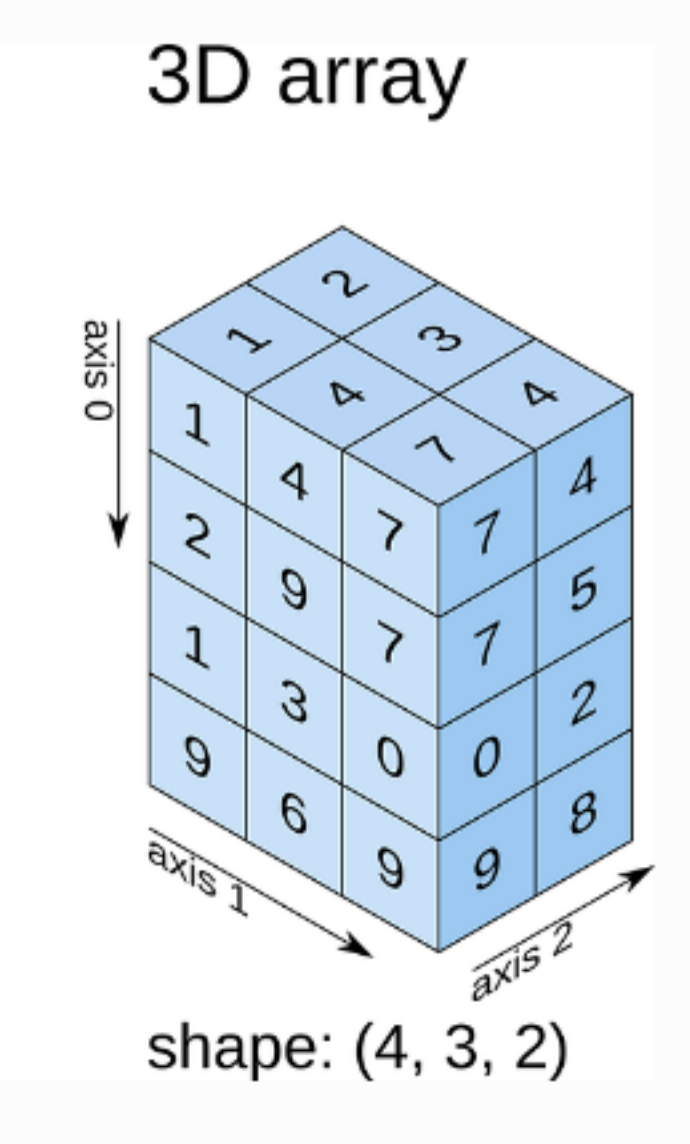
Axes

NUMPY AXES ARE THE DIRECTIONS ALONG THE ROWS AND COLUMNS

Just like coordinate systems, NumPy arrays also have axes.



Axes



Axes

Axes: directions along dimensions

Example

Some methods acts on arrays like lists (i.e. on all elements):

```
a=np.array([[ 0,  1,  2,  3],
            [ 4,  5,  6,  7],
            [ 8,  9, 10, 11]])
a.sum()
a.prod()
a.max()
a.min()
```

they may be used “across axes”…

along the rows, or the columns

Axes

```
b=np.arange(12).reshape(3,4)
        array([[ 0,  1,  2,  3],
               [ 4,  5,  6,  7],
               [ 8,  9, 10, 11]])
b.sum(axis=0)
        array([12, 15, 18, 21])
```

the sum is taken over the columns, “along axis 0”

```
b.sum(axis=1)
        array([ 6, 22, 38])
```

the sum is taken over the rows, “along axis 1”

Axes

More than 2 dims: “along axis” less intuitive
In general reduces the dimension (of 1, in this case)

```
c=np.arange(12).reshape(2,3,2)
          array([[[ 0,  1],
                  [ 2,  3],
                  [ 4,  5]],
                 [[ 6,  7],
                  [ 8,  9],
                  [10, 11]]])
c.shape   (2,3,2)
c0=c.sum(axis=0)
c0.shape       #we project along axis 0, so: (3,2)
c0
          array([[ 6,  8],
                 [10, 12],
                 [14, 16]])
```

Structural indexing tools: **ellipsis**

ellipsis

`...`

stands for as many : needed to reach x.ndim.

There may only be a single ellipsis present.

```
>>>x = np.array([[[1],[2],[3]],
[[4],[5],[6]]],[[[1],[2],[3]], [[4],[5],[6]]]])
>>> x.shape
(2, 2, 3, 1)
>>> x[0,...,0]          # same as x[0, : , : , 0]
array([[1, 2, 3],
       [4, 5, 6]])
```

Boolean operations

Standard boolean operators (and, or, not…) are *not vectorised*

(because they are not defined through __xxx__ methods… remember?)

```
>>> x
array([[21., 19., 18.,  3.],
       [ 3.,  4., 19., 21.]])
>>> not (x>4)
Traceback (most recent call last):
  File "<pyshell#330>", line 1, in <module>
    not (x>4)
ValueError: The truth value of an array with more than one element is ambiguous. Use a.any() or a.all()
>>> ~(x>4)
array([[False, False, False,  True],
       [ True,  True, False, False]])
```

Boolean operations

Some functions are redefined for this. Use:

`~` for `not`

`&` for `and`

`|` for `or`

```
>>> x
array([[21., 19., 18.,  3.],
       [ 3.,  4., 19., 21.]])
>>> (x>4) | (x==4)
array([[True, True, True, False],
       [False, True, True, True]])
```

Student Opinion Survey - for this Module1 of ILAI

UniBO-wide

To improve teaching and to detect strengths and weaknesses

Completely anonymous

- login necessary to be able to intercept absentees as well

I will not see the answers until the end of the academic year

Also available in Italian (top-right corner)

Student Opinion Survey - for this Module1 of ILAI

UniBO-wide

To improve teaching and to detect strengths and weaknesses

Completely anonymous

- login necessary to be able to intercept absentees as well

Also available in Italian (top-right corner)

Connect to:

http://val.unibo.it/

and use the code:

NumPy: broadcasting

some recipes

Structural indexing tools: **numpy.newaxis**

numpy.newaxis *create an axis of length one*

```
>>> x=np.array([1,2,3,4])          shape:    (4,)
>>> x[np.newaxis,:]
array([[1, 2, 3, 4]])              shape:    (1,4)
>>> x[np.newaxis,:].shape
(1, 4)
>>> x[:,np.newaxis]                shape:    (4,1)
array([[1],
       [2],
       [3],
       [4]])
```

newaxis is an alias for 'None', and 'None' can be used in place of this with the same result.

Structural indexing tools: **numpy.newaxis**

Example: create a matrix of the sums of two series

```
x=np.arange(6)
y=np.arange(0,60,10)

T= ???
```

```
T = array([[ 0,  1,  2,  3,  4,  5],
           [10, 11, 12, 13, 14, 15],
           [20, 21, 22, 23, 24, 25],
           [30, 31, 32, 33, 34, 35],
           [40, 41, 42, 43, 44, 45],
           [50, 51, 52, 53, 54, 55]])
```

I need 6 rows, 6 columns
Then broadcasting…

Structural indexing tools: **numpy.newaxis**

Example: create a matrix of the sums of two series

```
x=np.arange(6)
y=np.arange(0,60,10)

T= ???
```

```
T = array([[ 0,  1,  2,  3,  4,  5],
           [10, 11, 12, 13, 14, 15],
           [20, 21, 22, 23, 24, 25],
           [30, 31, 32, 33, 34, 35],
           [40, 41, 42, 43, 44, 45],
           [50, 51, 52, 53, 54, 55]])
```

I need 6 rows, 6 columns
Then *broadcasting*…

T = `y[:,np.newaxis]+x[np.newaxis,:]`

Structural indexing tools: **numpy.newaxis**

Example: create a matrix of the sums of two series

```
x=np.arange(6)
y=np.arange(0,60,10)

T= ???
```

```
T = array([[ 0,  1,  2,  3,  4,  5],
           [10, 11, 12, 13, 14, 15],
           [20, 21, 22, 23, 24, 25],
           [30, 31, 32, 33, 34, 35],
           [40, 41, 42, 43, 44, 45],
           [50, 51, 52, 53, 54, 55]])
```

I need 6 rows, 6 columns
Then *broadcasting*…

T = `y[:,np.newaxis]+x[np.newaxis,:]`

General recipe for tables of an operation (the "outer operation") on two series!

Broadcasting

Allow functions with inputs with different shape (sometimes… ☺)

```
>>> a=np.array([1])
>>> b=np.array([7,8,9])
>>> a+b
array([ 8,  9, 10])
```

shape: (1,)

shape: (3,)

shape: (3,)

Broadcasting

Allow functions with inputs with different shape (sometimes… ☺)

```
>>> a=np.array([1])
>>> b=np.array([7,8,9])
>>> a+b
array([ 8,  9, 10])
```

shape: (1,)

shape: (3,)

shape: (3,)

Same number of dimensions (both 1-D arrays)

A dimension with value 1 in `a`

is extended/stretched to match the corresponding dimension in `b`

The *single* value in that dimension is replicated

```
a: [1] ➔     [1, 1, 1]
b:           [7, 8, 9]
```

Broadcasting

Allow functions with inputs with different shape (sometimes… ☺)

```
>>> a=np.array([[1,2,3],[4,5,6]])
>>> b=np.array([7,8,9])
>>> a+b
array([[ 8, 10, 12],
       [11, 13, 15]])
```

shape: (2,3)

shape: (3,)

shape: (2,3)

Broadcasting

Allow functions with inputs with different shape (sometimes… ☺)

```
>>> a=np.array([[1,2,3],[4,5,6]])
>>> b=np.array([7,8,9])
>>> a+b
array([[ 8, 10, 12],
       [11, 13, 15]])
```

shape: (2,3)

shape: (3,)

shape: (2,3)

Different dimensions

The missing dimension in b is added, as 1 *leading*

b.shape: (3,) ➔ (1,3) b: `[7,8,9]` ➔ `[[7,8,9]]`

Now we are in "same dimensions" a: (2,3) b: (1,3)

The *single* value in the dimension 1 is replicated to match the other

```
b: [[7,8,9]] ➔   [[7,8,9],
                  [7,8,9]]
```

Broadcasting

Remember:

shape:	a tuple (d1, d2, ... , dn)
dimension(s):	len(shape): n
size of dimension k:	dk

Two rules of broadcasting:

(1) if one argument has *less dimensions*

add as many leading 1s (i.e. on left) as possible to the shape

(2) if one argument has size 1 on dimension k

stretch that dimension to match the dimension of the other array on that dimension

After application of the rules, *the sizes of all arrays must match*

Broadcasting

Exercise:

```
A (3d array):          15 x 3 x 5
B (3d array):          15 x 1 x 5
Result (3 d array):    15 x 3 x 5

A (3d array):          15 x 3 x 5
B (2d array):             3 x 1
Result (3 d array):    15 x 3 x 5

A (2d array):             2 x 1
B (3d array):          8 x 4 x 3
Result (? d array):
```

Broadcasting

Exercise:

```
A (3d array):          15 x 3 x 5
B (3d array):          15 x 1 x 5
Result (3 d array):    15 x 3 x 5

A (3d array):          15 x 3 x 5
B (2d array):           1 x 3 x 1
Result (3 d array):    15 x 3 x 5

A (2d array):          1 x 2 x 1
B (3d array):          8 x 4 x 3
Result (? d array): IMPOSSIBLE
```

Example: outer sum

```
x=np.arange(6)
y=np.arange(0,60,10)
T = y[:,np.newaxis]+x[np.newaxis,:]
```

could be written also as

```
T = y[:,np.newaxis]+x
```

```
T = array([[ 0,  1,  2,  3,  4,  5],
           [10, 11, 12, 13, 14, 15],
           [20, 21, 22, 23, 24, 25],
           [30, 31, 32, 33, 34, 35],
           [40, 41, 42, 43, 44, 45],
           [50, 51, 52, 53, 54, 55]])
```

General recipe for tables of an operation (the "outer operation") on two series!

Example: outer sum

```
x=np.arange(6)
y=np.arange(0,60,10)
T = y[:,np.newaxis]+x[np.newaxis,:]
```

could be written also as

```
T = y[:,np.newaxis]+x
```

```
x.shape                          (6,)
y.shape                          (6,)
y[:,np.newaxis].shape          (6,1,)
[0,10,20,30,40,50] ➔ [[0],
                      [10],
                      [20],
                      [30],
                      [40],
                      [50]])
```

By rule (1), `x.shape` ➔ `(1,6,)`

`[0,1,2,3,4,5]` ➔ `[[0,1,2,3,4,5]]`

Now we apply rule (2) twice, once on x once on y

Example: outer sum

```
y[:,np.newaxis].shape        (6,1,)
[0,10,20,30,40,50] ➔ [ [0],
                       [10],
                       [20],
                       [30],
                       [40],
                       [50]])
```

By rule (1), `x.shape` ➔ `(1,6,)`

`[0,1,2,3,4,5]` ➔ `[[0,1,2,3,4,5]]`

Now we apply rule (2) twice, once on x once on y

```
y[:,np.newaxis].shape        (6,1,) ➔      (6,6)
[0,10,20,30,40,50] ➔ [ [0],    ➔ [[ 0,  0,  0,  0,  0,  0],
                       [10],       [10, 10, 10, 10, 10, 10],
                       [20],       [20, 20, 20, 20, 20, 20],
                       [30],       [30, 30, 30, 30, 30, 30],
                       [40],       [40, 40, 40, 40, 40, 40],
                       [50]]       [50, 50, 50, 50, 50, 50]]
```

Example: outer sum

```
y[:,np.newaxis].shape        (6,1,)
[0,10,20,30,40,50] ➔ [ [0],
                       [10],
                       [20],
                       [30],
                       [40],
                       [50]])
```

By rule (1), `x.shape` ➔ `(1,6,)`

`[0,1,2,3,4,5]` ➔ `[[0,1,2,3,4,5]]`

Now we apply rule (2) twice, once on x once on y

`x.shape` ➔ `(1,6,)` ➔ `(6,6)`

`[0,1,2,3,4,5]` ➔ `[[0,1,2,3,4,5]]` ➔

```
[[0,1,2,3,4,5],
 [0,1,2,3,4,5],
 [0,1,2,3,4,5],
 [0,1,2,3,4,5],
 [0,1,2,3,4,5]]
```

Example: outer sum

```
y                    ➔ y[:,np.newaxis]

[0,10,20,30,40,50] ➔ [ [0],    ➔ [[ 0,  0,  0,  0,  0,  0],
                       [10],       [10, 10, 10, 10, 10, 10],
                       [20],       [20, 20, 20, 20, 20, 20],
                       [30],       [30, 30, 30, 30, 30, 30],
                       [40],       [40, 40, 40, 40, 40, 40],
                       [50]]       [50, 50, 50, 50, 50, 50]]

x
[0,1,2,3,4,5] ➔[[0,1,2,3,4,5]] ➔[[ 0,  1,  2,  3,  4,  5],
                                 [ 0,  1,  2,  3,  4,  5],
                                 [ 0,  1,  2,  3,  4,  5],
                                 [ 0,  1,  2,  3,  4,  5],
                                 [ 0,  1,  2,  3,  4,  5],
                                 [ 0,  1,  2,  3,  4,  5]]
```

Now just apply the *universal* (vectorised) +

Some functionalities

Concatenate

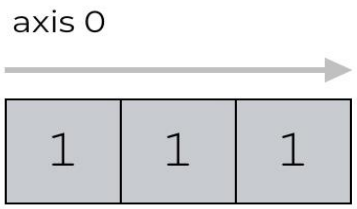
```
Ones  array([[1, 1, 1],
             [1, 1, 1]])
Twos  array([[2, 2, 2],
             [2, 2, 2]])

Conc_over0 = np.concatenate([Ones, Twos], axis = 0)
Conc_over0:  [[1 1 1]
              [1 1 1]
              [2 2 2]
             [2 2 2]]

Conc_over1 = np.concatenate([Ones, Twos], axis = 1)
Conc_over1:  [[1 1 1 2 2 2]
              [1 1 1 2 2 2]]
```

Concatenate on 1D arrays

In a 1-d array, there is only one axis



An attempt to concatenate two 1-D arrays along axis 1 is an error:

IndexError: axis 1 out of bounds [0, 1)

Shape Manipulation

All these gives views

```
a.ravel()        # returns the array, flattened
a.reshape(6,2)   # modified shape
a.T              # returns the array, transposed
```

In ravel the order is “row-major, C-style order” (the rightmost index “changes the fastest”, so the element after a[0,0] is a[0,1]).

For use column-major, Fortran-style order, use a.ravel(order='F')

Transpose (and array vs vectors)

Given an array X with shape `(d1,d2,…,dn)`

its transpose has shape `(dn,…,d2,d1)`

`X.T` is the *transpose* of `X` (it is a view on `X`)

`X.T[i1,…,in]` is `X[in,…,i1]`

Let X be a 1D array: `X.shape= (k,)`

Who is `X.T` ?

Transpose (and array vs vectors)

Given an array X with shape `(d1,d2,…,dn)`

its transpose has shape `(dn,…,d2,d1)`

`X.T` is the *transpose* of `X` (it is a view on `X`)

`X.T[i1,…,in]` is `X[in,…,i1]`

Let X be a 1D array: `X.shape= (k,)`

Who is `X.T` ?

The same as `X`!

shape `(k,)` inverted is always `(k,)`

Transpose (and array vs vectors)

Especially delicate if taking rows (columns) as slices of 2-D arrays:

```
>>> M=np.arange(1,10).reshape(3,3)
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])
>>> M[:,0]
array([1, 4, 7])
>>> M[0,:]
array([1, 2, 3])
# same shape: (3,)
```

Shape Manipulation

Instead, the **method** ndarray.resize *modifies the array itself:*

```
>>> a=np.array([[ 2.,  8.,  0.,  6.],
                [ 4.,  5.,  1.,  1.],
                [ 8.,  9.,  3.,  6.]])
>>> a.resize((2,6))
>>> a
array([[ 2.,  8.,  0.,  6.,  4.,  5.],
       [ 1.,  1.,  8.,  9.,  3.,  6.]])
```

`np.random.random()`

Return random floats in the half-open interval [0.0, 1.0).

`np.random.random(`*`shape`*`)`

return an array of random number of the given shape

Stacking together different arrays

Several arrays can be stacked together along different axes:

```
>>> a = np.floor(10*np.random.random((2,2)))
>>> a
array([[ 8.,  8.],
       [ 0.,  0.]])
>>> b = np.floor(10*np.random.random((2,2)))
>>> b
array([[ 1.,  8.],
       [ 0.,  4.]])
>>> np.vstack((a,b))
array([[ 8.,  8.],
       [ 0.,  0.],
       [ 1.,  8.],
       [ 0.,  4.]])
>>> np.hstack((a,b))
array([[ 8.,  8.,  1.,  8.],
       [ 0.,  0.,  0.,  4.]])
```

Stacking together different arrays

vstack is equivalent to concatenate on axis=0

… but expands 1D arrays to 2D (shape (N,) into (1, N))

hstack instead calls concatenate with axis=1

Splitting arrays into smaller pieces

```
>>> a = np.floor(10*np.random.random((2,12)))
>>> a
array([[ 9.,  5.,  6.,  3.,  6.,  8.,  0.,  7.,  9.,  7.,  2.,  7.],
       [ 1.,  4.,  9.,  2.,  2.,  1.,  0.,  6.,  2.,  2.,  4.,  0.]])
>>> np.hsplit(a,3)   # Split a into 3
[array([[ 9.,  5.,  6.,  3.],
        [ 1.,  4.,  9.,  2.]]),
 array([[ 6.,  8.,  0.,  7.],
        [ 2.,  1.,  0.,  6.]]),
 array([[ 9.,  7.,  2.,  7.],
        [ 2.,  2.,  4.,  0.]])]
>>> np.hsplit(a,(3,4))   # Split a after the third and the fourth column
[array([[ 9.,  5.,  6.],
        [ 1.,  4.,  9.]]),
 array([[ 3.],
        [ 2.]]),
 array([[ 6.,  8.,  0.,  7.,  9.,  7.,  2.,  7.],
        [ 2.,  1.,  0.,  6.,  2.,  2.,  4.,  0.]])]
```

Splitting arrays into smaller pieces

vsplit splits along the vertical axis, and array_split allows one to specify along which axis to split.

Returning the indices to access the main diagonal of an array

numpy.diag_indices(*n*, *ndim=2*)

array a with a.ndim >= 2 dimensions and shape (n, n, …, n)

```
>>> di = np.diag_indices(4)
>>> di
(array([0, 1, 2, 3]), array([0, 1, 2, 3]))
>>> a = np.arange(16).reshape(4, 4)
>>> a[di] = 100
>>> a
array([[100,   1,   2,   3],
       [  4, 100,   6,   7],
       [  8,   9, 100,  11],
       [ 12,  13,  14, 100]])


>>> sum(a[di])
400
```

Returning the indices to access the main diagonal of an array

Filling the diagonal may be obtained more efficiently via

numpy.fill_diagonal(*a*, *val*)

Fill the main diagonal of the given array of any dimensionality:

np.fill_diagonal(a,100)

fromfunction and if statement

We saw that `np.fromfunction` allows the costruction of arrays by mapping a function (passed as argument to `np.fromfunction`) to an input array of a certain shape:

```
import numpy as np
def f(x,y):
    return x+y
np.fromfunction(f,(3,3),dtype=int)
```

returns

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

fromfunction and if statement

However a program like the following

```
def f(x,y):
    if x==y:
        return 0
A=np.fromfunction(f,(2,2),dtype=int)
```

returns an error.

The reason is the use of the conditional command inside the function (`f`) passed to `fromfunction`

fromfunction and if statement

The reason of the error is the use of the conditional command inside the function (`f`) passed to `fromfunction`

The implementation of `fromfunction`, in fact, applies the *vectorization* of the operations used inside its parameter (`f`).

In this way it results in a very efficient (lower level) iteration of `f` over the elements of its argument arrays

If inside `f` we use non vectorizable operations (like the boolean operations, or the conditional `if`), we get an error.

A workaround is to explicitly "vectorize" the function…

fromfunction and if statement

If inside `f` we use non vectorizable operations (like the boolean operations, or the conditional `if`), we get an error. Example: in

```
def f(x,y):
    if x==y:
        return 0
A=np.fromfunction(f,(2,2),dtype=int)
```

in `fromfunction` the guard `x==y` will evaluate to

```
array([[ True, False],
       [False,  True]])
```

which is not a valid truth value for the guard of an `if`

fromfunction and if statement

A workaround is to explicitly "vectorize" the function. Instead of passing f to `fromfunction`, pass instead `np.vectorize(f)`:

```
def f(x,y):
    if x==y:
        return 0
A=np.fromfunction(np.vectorize(f), (2,2), dtype=int)
```

`np.vectorize(f)` will use for loops to explicitly apply `f` to the elements of the input array, without using the built-in vectorization of the predefined operations.

It is slower, of course. Not the intended use.

On Panopto (Virtuale, right column)

A video of prof Lodi and prof Martini

solving and discussing

some of the lab exercises on Numpy

ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

End of the Module 1. Thank you! ☺

Michael Lodi

(all course slide credits: Simone Martini)

michael.lodi@unibo.it