Introduction to Programming Lecture 10

- Instructor : Andrew Yarmola andrew.yarmola@uni.lu
- Course Schedule : Wednesday 14h00 - 15h30 Campus Kirchberg B21
- Course Website : sites.google.com/site/andrewyarmola/itp-uni-lux
- Office Hours : Thursday 16h00 - 17h00 Campus Kirchberg G103 and by appointment.

Project 1 on GitHub by noon Friday Nov 25

Feel free to turn in Homework 9 on Monday Nov 28

`numpy`

`numpy` is a module of python that focuses on multi-dimensional arrays, scientific computation, and high efficiency. Combined with `scipy`, `sympy`, and graphing modules, one can use python in highly customizable scientific computing.

`array` object

`numpy` is known for *array oriented computing*. An `array` is a mutli-dimensional table (in dimension 2, you can think of an `array` as a matrix.

In [1]:

```
import numpy as np

# we can make an array from a list of lists
a = np.array([[0,1,2],[3,4,5]]) # 2-d
b = np.array([[[0,1],[2,3]],[[4,5],[6,7]]]) # 3-d
```

In [2]:

```
a
```

Out[2]:

```
array([[0, 1, 2],
       [3, 4, 5]])
```

In [3]:

```
b
```

Out[3]:

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

       [[4, 5],
        [6, 7]]])
```

In [4]:

```
a.ndim
```

Out[4]:

```
2
```

In [5]:

```
b.ndim
```

Out[5]:

```
3
```

In [6]:

```
a.shape
```

Out[6]:

```
(2, 3)
```

In [7]:

```
b.shape
```

Out[7]:

```
(2, 2, 2)
```

Basic indexing, assignment, and copies

To get values out of an array, we use the syntax `array[p_1,p_2,p_3,...]` where `p_i` is the index in that dimension (also known as **axis**) starting with zero.

In [8]:

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

Out[8]:

```
2
```

In [9]:

```
print(repr(a))

a[1,1] = 29

print(repr(a))
```

```
array([[0, 1, 2],
       [3, 4, 5]])
array([[ 0,  1,  2],
       [ 3, 29,  5]])
```

In [10]:

```
# make a copy of a
c = a.copy()
c[0,0] = 9
print(repr(c))
```

```
array([[ 9,  1,  2],
       [ 3, 29,  5]])
```

We will look at array slicing towards the end of this lecture.

Getting help

Along with the standrad `help()` and `?` methods of looking up documentation in `ipython`, we can also search the documentation using `numpy.lookfor()`. This can be useful if you can't quite remember the exact name of a command.

In [11]:

```
np.lookfor('evenly spaced values')
```

```
Search results for 'evenly spaced values'
-----------------------------------------
numpy.arange
    Return evenly spaced values within a given interval.
numpy.ma.arange
    Return evenly spaced values within a given interval.
numpy.logspace
    Return numbers spaced evenly on a log scale.
numpy.geomspace
    Return numbers spaced evenly on a log scale (a geometric progr
ession).
numpy.trapz
    Integrate along the given axis using the composite trapezoidal
rule.
numpy.gradient
    Return the gradient of an N-dimensional array.
```

Efficiency

At first glance, `arrays` look just like lists of lists, however, they are **much more efficient** (for certain tasks)

In [12]:

```
# we can time basic creation
%timeit list(range(1000))
```

```
17.1 µs ± 69.5 ns per loop (mean ± std. dev. of 7 runs, 10000 loop
s each)
```

In [13]:

```
# `numpy` not only creates faster, it can be smart
%timeit np.arange(1000)
```

```
1.24 µs ± 125 ns per loop (mean ± std. dev. of 7 runs, 1000000 loo
ps each)
```

In [14]:

```
# operations are even slower
%timeit [i**2 for i in range(1000)]
```

```
373 µs ± 78.5 µs per loop (mean ± std. dev. of 7 runs, 1000 loops
each)
```

In [15]:

```
# squares are even slower
%timeit np.arange(1000)**2
```

```
2.93 µs ± 8.56 ns per loop (mean ± std. dev. of 7 runs, 100000 loo
ps each)
```

Creating `arrays`

As we just saw above, it is much faster to create arrays directly than to turn lists into arrays. `numpy` has many build in methods for array creation. Here are a few :

In [16]:

```
# evenly spaced
a = np.arange(2,10,2)
print(repr(a))
```

```
array([2, 4, 6, 8])
```

In [17]:

```
# evenly spaced by number of points
a = np.linspace(2,7,10)
print(repr(a))
```

```
array([2.        , 2.55555556, 3.11111111, 3.66666667, 4.22222222,
       4.77777778, 5.33333333, 5.88888889, 6.44444444, 7.        ]
)
```

In [18]:

```
# zeros of a given shape
a = np.zeros((4,2,3))
print(repr(a))
```

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

       [[0., 0., 0.],
        [0., 0., 0.]],

       [[0., 0., 0.],
        [0., 0., 0.]],

       [[0., 0., 0.],
        [0., 0., 0.]]])
```

In [19]:

```
# ones of a given shape
a = np.ones((2,3,4))
print(repr(a))
```

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

       [[1., 1., 1., 1.],
        [1., 1., 1., 1.],
        [1., 1., 1., 1.]]])
```

In [20]:

```
# identity matrix
a = np.eye(4)
print(repr(a))
```

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

In [21]:

```
# diagonal matrix
d = np.linspace(0,1,4)
a = np.diag(d) # the argument can be any iterable
print(repr(a))
```

```
array([[0.        , 0.        , 0.        , 0.        ],
       [0.        , 0.33333333, 0.        , 0.        ],
       [0.        , 0.        , 0.66666667, 0.        ],
       [0.        , 0.        , 0.        , 1.        ]])
```

In [22]:

```
# random arrays
# uniform in [0,1]
a = np.random.rand(3,1)
print(repr(a))
```

```
array([[0.50755507],
       [0.0211933 ],
       [0.43352176]])
```

In [23]:

```
shape = (2,1,3)
# standard normal distribution (Gaussian)
a = np.random.randn(*shape)
print(repr(a))
```

```
array([[[-1.55334234, -0.3192978 ,  0.52704645]],

       [[ 0.7111124 , -0.21754548,  2.63779121]]])
```

Remark. `numpy` has a **much better** random number generator in `numpy.random`

If you every need to **claim** space for an array quickly and with **arbitrary** (but not random) values, you can use `numpy.empty`

In [24]:

```
# "empty" array of a given shape
a = np.empty((3,1,4))
print(repr(a))
```

```
array([[[2.31584178e+077, 2.31584178e+077, 7.41098469e-323,
         0.00000000e+000]],

       [[0.00000000e+000, 0.00000000e+000, 0.00000000e+000,
         0.00000000e+000]],

       [[0.00000000e+000, 0.00000000e+000, 0.00000000e+000,
         0.00000000e+000]]])
```

Warning. You have no guarantee about the values of `numpy.empty`

In [28]:

```
# "empty" array of another array
b = np.empty_like(a)
print(repr(b))
```

```
array([[[2.31584178e+077, 2.31584178e+077, 7.41098469e-323,
         0.00000000e+000]],

       [[0.00000000e+000, 0.00000000e+000, 0.00000000e+000,
         0.00000000e+000]],

       [[0.00000000e+000, 0.00000000e+000, 0.00000000e+000,
         0.00000000e+000]]])
```

Remark. Note that most methods take a `shape` as a tuple, however, some methods such as `numpy.random.randint` expect the `shape` as positional arguments.

Data types

Unlike python `lists` (or lists of lists), `numpy arrays` can **only store one type of value** at a time. Here is an example :

In [29]:

```
a = np.arange(4)
print(repr(a))
```

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

In [30]:

```
a[2] = 5 # change value at 2
print(repr(a))
```

```
array([0, 1, 5, 3])
```

In [31]:

```
a[1] = 7.6 # try to set a float
print(repr(a))
```

```
array([0, 7, 5, 3])
```

From these examples, we can see the following things :

- arrays are mutable types
- for 1-dim arrays, we can access just like a list (more on this alter)
- **when we tried to set a `float` for `a[1]`, the array turned it into an `int`**

Why did this happen? It turns out that when our array was created, its data type (`dtype`) was set to being an integer.

In [32]:

```
a.dtype
```

Out[32]:

```
dtype('int64')
```

In [33]:

```
b = np.array([1.,2.,3.,4.])
b.dtype
```

Out[33]:

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

In [34]:

```
c = np.array('a list of words'.split())
c.dtype
```

Out[34]:

```
dtype('<U5')
```

Remark : Above, `'int64'` means integer using at most `64` bits. The `'<U5'` type stands for 5 or less **unicode** characters.

In [35]:

```
# the type won't change when I set a value
c[0] = 'something'
print(repr(c))
```

```
array(['somet', 'list', 'of', 'words'], dtype='<U5')
```

In [36]:

```
# a copy with different type
d = c.astype('<U10')
print(repr(d))

d[0] = 'something'
print(repr(d))
```

```
array(['somet', 'list', 'of', 'words'], dtype='<U10')
array(['something', 'list', 'of', 'words'], dtype='<U10')
```

Structured data types

If you want to store types a little more complicate than just strings, integers, floating point and complex numbers, you can use **structured data types**

In [37]:

```
samples = np.array([('ALFA',1,0.37),
                    ('BETA',2,0.11),
                    ('TAU',1,0.13)])
print(samples.shape)
print(repr(samples))
```

```
(3, 3)
array([['ALFA', '1', '0.37'],
       ['BETA', '2', '0.11'],
       ['TAU', '1', '0.13']], dtype='<U4')
```

In [38]:

```
# data type spec
t = np.dtype([('sensor_code', '<U4'),
              ('position', 'int64'),
              ('value', 'float64')])
```

In [39]:

```
samples = np.array([[('ALFA',1,0.37)],
                    [('BETA',2,0.11)],
                    [('TAU',1,0.13)]], dtype=t)
print(samples.shape)
print(repr(samples))
```

```
(3, 1)
array([[('ALFA', 1, 0.37)],
       [('BETA', 2, 0.11)],
       [('TAU', 1, 0.13)]],
      dtype=[('sensor_code', '<U4'), ('position', '<i8'), ('value'
, '<f8')])
```

In [40]:

```
# we can use the type name to
# get a nice subarray
samples['sensor_code']
```

Out[40]:

```
array([['ALFA'],
       ['BETA'],
       ['TAU']], dtype='<U4')
```

In [41]:

```
# we can also make assignemtns
samples[0]['sensor_code'] = 'XI'
print(repr(samples))
```

```
array([[('XI', 1, 0.37)],
       [('BETA', 2, 0.11)],
       [('TAU', 1, 0.13)]],
      dtype=[('sensor_code', '<U4'), ('position', '<i8'), ('value'
, '<f8')])
```

Remark. It is possible to write your own classes that can be used as data types for `numpy` arrays, but this is rather difficult.

Basic operatons

`numpy arrays` behave **very differently** from python `lists` when it comes to binary operations such as `+,-,*,/,**,%, ==, <, >, etc`. All of the standard operations are **element-wise**

In [62]:

```
a = np.array([[1,-1,2],[3,4,5]])
b = np.array([[3,0,6],[6,4,5]])
print(repr(a))
print(repr(b))
```

```
array([[ 1, -1,  2],
       [ 3,  4,  5]])
array([[3, 0, 6],
       [6, 4, 5]])
```

In [63]:

```
# element-wise addition
a+b
```

Out[63]:

```
array([[ 4, -1,  8],
       [ 9,  8, 10]])
```

In [64]:

```
# element-wise multiplication
a*b
```

Out[64]:

```
array([[ 3,  0, 12],
       [18, 16, 25]])
```

Warning : For 2-dim arrays, `a*b` **is not matrix multiplication**! To multiply matrices, use `a.dot(b)`. We will talk about linear algebra a little later.

In [65]:

```
# element-wise division
(b*a+a)/a
```

Out[65]:

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

In [66]:

```
# element-wise exp
a**b
```

Out[66]:

```
array([[   1,    1,   64],
       [ 729,  256, 3125]])
```

In [67]:

```
# elemnt-wise remainder
a % b
```

Out[67]:

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

For operations with integers and floats everything is still element-wise

In [68]:

```
print(repr(a))
a + 1.5
```

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

Out[68]:

```
array([[2.5, 0.5, 3.5],
       [4.5, 5.5, 6.5]])
```

In [69]:

```
3/a
```

Out[69]:

```
array([[ 3.  , -3.  ,  1.5 ],
       [ 1.  ,  0.75,  0.6 ]])
```

In [70]:

```
a**2
```

Out[70]:

```
array([[ 1,  1,  4],
       [ 9, 16, 25]])
```

In [71]:

```
a % 3
```

Out[71]:

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

You can also do **sums and products** of all elements of just along an axis. You can use both `numpy.prod()` and`numpy.sum()` functions or use instance methods `.prod()` and `.sum()`.

In [72]:

```
# sum all elements
print(np.sum(a))

# product of all elements
print(a.prod())
```

```
14
-120
```

In [73]:

```
# sum along the 0-axis (rows)
a.sum(axis = 0)
```

Out[73]:

```
array([4, 3, 7])
```

In [74]:

```
# product along the 1-axis (cols)
np.prod(a, axis = 1)
```

Out[74]:

```
array([-2, 60])
```

Comparisons are also element-wise

In [75]:

```
a == 5
```

Out[75]:

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

In [76]:

```
a > 3
```

Out[76]:

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

In [77]:

```
# element-wise comaprison
a == b
```

Out[77]:

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

In [78]:

```
b < a
```

Out[78]:

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

In [79]:

```
# to check if arrays are equal
np.array_equal(a,b)
```

Out[79]:

```
False
```

To check if all elements of a boolean array are `True`, you can use either `numpy.all()` or the `.all()` instance method. Similar for `any`.

In [80]:

```
c = (a <= a**b)
c.all()
```

Out[80]:

```
True
```

Vectorized functions and vectorization

Most mathematical functions implemented in `numpy` act element-wise on arrays. Vectorization is the process of turning a function into one that works element-wise on arrays. For example,

In [81]:

```
np.log(a)
```

Out[81]:

```
array([[0.        ,        nan, 0.69314718],
       [1.09861229, 1.38629436, 1.60943791]])
```

You can create your own element-wise functions by using `numpy.vectorize`.

Example 1

In [82]:

```
def bad_inv(x) :
    if x < 1 : return x**3
    else : return 1/x
```

In [83]:

```
bad_inv(a)
```

```
-----------------------------------------------------------------------
---------
ValueError                                Traceback (most recent c
all last)
<ipython-input-83-8c1d86565fcb> in <module>()
----> 1 bad_inv(a)

<ipython-input-82-b1fcdf8d20f5> in bad_inv(x)
      1 def bad_inv(x) :
----> 2     if x < 1 : return x**3
      3     else : return 1/x

ValueError: The truth value of an array with more than one element
is ambiguous. Use a.any() or a.all()
```

In [84]:

```
# vectorize
v_bad_inv = np.vectorize(bad_inv)
v_bad_inv(a)
```

Out[84]:

```
array([[ 1.        , -1.        ,  0.5       ],
       [ 0.33333333,  0.25      ,  0.2       ]])
```

Example 2

In [85]:

```
def silly_sum(x,y) :
    if x < y : return x + 2*y
    else : return x - 17*y

v_silly_sum = np.vectorize(silly_sum)
v_silly_sum(b,a)
```

Out[85]:

```
array([[-14,  17, -28],
       [-45, -64, -80]])
```

Example 3

In [86]:

```
from functools import partial
def on_ec(a,b,x,y) :
    return y**2 == x**3 + a*x + b

v_on_ec = np.vectorize(partial(on_ec,1,1))
v_on_ec(b,a)
```

Out[86]:

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

Broadcasting

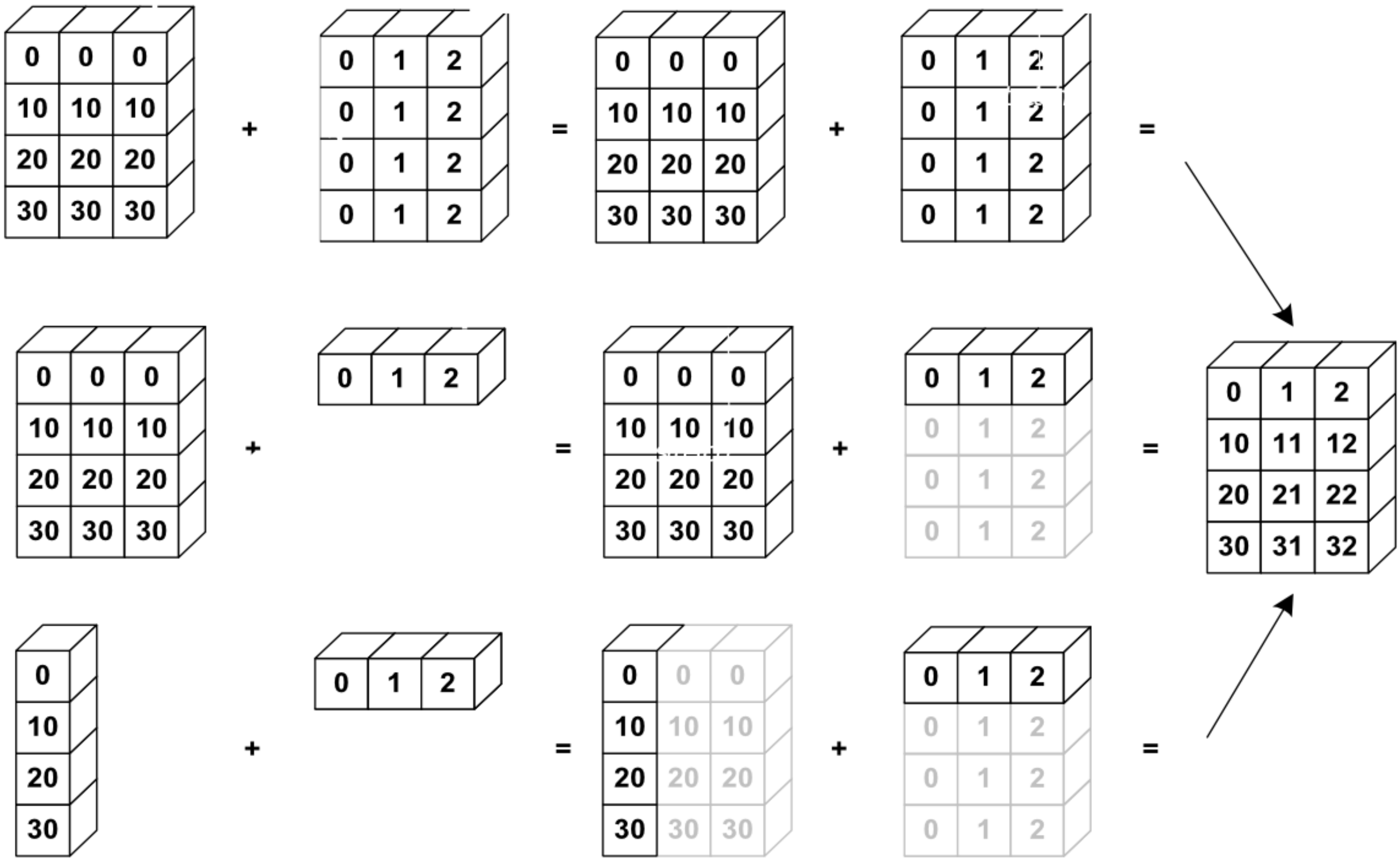
`numpy` is very clever about how it perfoms binary operations on arrays of difference shapes and dimenions. Here is an example :

In [87]:

```
a = np.ones((3,3))
b = np.array([2,3,4])
print(repr(a))
print(repr(b))
print('+'*25)
print(repr(a+b))
```

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

Broadcasting describes how `numpy` works with arrays of different shapes during arithmetic operation. Subject to certain constraints, the smaller array is “broadcast” across the larger array so that they have compatible shapes.



When operating on two arrays `a` and `b`, `numpy` compares their shape tuples for compatibility. Let `a.shape` $= (s_n, s_{n-1}, \ldots, s_0)$ and `b.shape` $= (t_m, t_{m-1}, \ldots, t_0)$. We set $k = max(m, n)$ and if $n < m$, set $s_{n+j} = 1$ or if $m > n$, set $t_{m+j} = 1$ for $1 \leq j \leq |m - n|$. We call this **padding** a shaphe on the left.

The (padded) shapes are **compatible** if for all $0 \leq i \leq k$ one has

- $s_i = t_i$ **or**
- one of s_i, t_i is equal to 1.

Notice that we compare from right to left.

Once the shapes are deemed compatible, `numpy` **broadcasts** each array as follows :

- if $s_i = 1 < t_i$, then `numpy` makes t_i copies of `a` and **stacks** them along the axis for s_i.
- similarly for `b` wherever $b_i = 1$.

Once each array has been broadcasts, they both have the shape $(\max(s_k, t_k), \ldots, \max(s_0, t_0))$.

For example,

In [88]:

```
a = np.arange(4)
b = np.array([[4],[7],[9]])
print(repr(a))
print(repr(b))
```

```
array([0, 1, 2, 3])
array([[4],
       [7],
       [9]])
```

In [89]:

```
print(a.shape)
print(b.shape)
```

```
(4,)
(3, 1)
```

We now pad `a.shape` on the left to obtain

```
a.shape = (1,4)
b.shape = (3,1)
```

We then take 3 copies of `a` and stack them along the 0-axis (rows) and 4 copies of `b` and stack them along the 1-axis (columns). See the image above.

In [90]:

```
c = a*b
print(repr(c))
print(c.shape)
```

```
array([[ 0,  4,  8, 12],
       [ 0,  7, 14, 21],
       [ 0,  9, 18, 27]])
(3, 4)
```

Remark : Your vectorized functions will work with broadcasting!

In [91]:

```
v_silly_sum(a,b)
```

Out[91]:

```
array([[ 8,  9, 10, 11],
       [14, 15, 16, 17],
       [18, 19, 20, 21]])
```

Repeat, tile, broadcast_to, and kron

There are several ways of using the data from an array to create new arrays by repetition.

- `np.broadcast_to(A, new_shape)` for a **compatible** shape, this returns a **readonly** version of `A` broadcast to a new shape

In [92]:

```
print(repr(a))

x = np.broadcast_to(a,(2,4))

print(repr(x))
```

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

- `np.repeat(A, repeats, axis = axis_num)` returns a new array with entries of `A` repeated along the axis `axis_num` according to the contents of `repeats`
  - `repeats` is an int or array of ints giving the number of repetitions for each element on the axis

In [93]:

```
print(repr(c))

y = np.repeat(c, 2, axis = 1)

print(repr(y))
```

```
array([[ 0,  4,  8, 12],
       [ 0,  7, 14, 21],
       [ 0,  9, 18, 27]])
array([[ 0,  0,  4,  4,  8,  8, 12, 12],
       [ 0,  0,  7,  7, 14, 14, 21, 21],
       [ 0,  0,  9,  9, 18, 18, 27, 27]])
```

In [94]:

```
# notice the 0 in [0,3,1] makes that
# row vanish
z = np.repeat(c, [0,3,1], axis = 0)
print(repr(z))
```

```
array([[ 0,  7, 14, 21],
       [ 0,  7, 14, 21],
       [ 0,  7, 14, 21],
       [ 0,  9, 18, 27]])
```

- `numpy.tile(A, reps)` construct an array by repeating `A` the number of times given by `reps` in each dimension/axis.
  - `reps` is an **list**/array giving the number of repetitions of A along each axis.
  - if `reps` has length `d`, the result will have dimension of `max(d, A.ndim)`.
  - if `A.ndim < d,` A is promoted to be d-dimensional by prepending new axes. So a shape (3,) array is promoted to (1, 3) for 2-D replication, or shape (1, 1, 3) for 3-D replication.
  - if A.ndim > d, `reps` is promoted to length `A.ndim` by prepending 1's to it. Thus for an `A` of shape `(2, 3, 4, 5)`, a reps of `(2, 2)` is treated as `(1, 1, 2, 2)`.

In [95]:

```
print(repr(b))

# double the rows and
# triple the columns
q = np.tile(b,(2,3))

print(repr(q))
```

```
array([[4],
       [7],
       [9]])
array([[4, 4, 4],
       [7, 7, 7],
       [9, 9, 9],
       [4, 4, 4],
       [7, 7, 7],
       [9, 9, 9]])
```

- `numpy.kron(a,b)` returns the **Kronecker product** : a composite array made of blocks of the second array scaled by the first.
  - for 2-dim you get something like :

```
[[ a[0,0]*b,   a[0,1]*b,  ... , a[0,-1]*b  ],
[   ...                              ...   ],
[ a[-1,0]*b,  a[-1,1]*b, ... , a[-1,-1]*b ]]
```

In [96]:

```
p = np.array([[1,3],[5,4]])
d = np.diag([1,2,3])

r = np.kron(d, p)

print(repr(r))
```

```
array([[ 1,  3,  0,  0,  0,  0],
       [ 5,  4,  0,  0,  0,  0],
       [ 0,  0,  2,  6,  0,  0],
       [ 0,  0, 10,  8,  0,  0],
       [ 0,  0,  0,  0,  3,  9],
       [ 0,  0,  0,  0, 15, 12]])
```

Reshaping

`ravel`

It is possible reshape array data to your liking. For example, you can **(un)`ravel`** (or **flatten**) a multi-dimensional array into a one dimensional array. Similar to how you would read off the entries of a matrix.

In [97]:

```
p = np.array([[1,2,3],[4,5,6]]) + np.array([[[0]],[[6]]])
print(repr(p))
# read off the entries in lexographic order
r = p.ravel()
print(repr(r))
```

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

       [[ 7,  8,  9],
        [10, 11, 12]]])
array([ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12])
```

The order here is **lexographic order** on the indices. In the above example, the order is

```
(0,0,0)
(0,0,1)
(0,0,2)
(0,1,0)
(0,1,1)
...
```

`reshape`

You can **reshape** an `array` by using

- `A.reshape(new_shape)`
- `numpy.reshape(A, new_shape)`

Note that `np.prod(A.shape)` must equal to `np.prod(new_shape)`. Both of there methods **return a "new" array object**.

In [98]:

```
p.reshape((3,1,4))
```

Out[98]:

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

       [[ 5,  6,  7,  8]],

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

Think of the above operation as **unraveling followed by placing the entries into the new shape in order**.

You can have `numpy` **intuit the rest of `new_shape`** by terminating the shape with `-1`

In [99]:

```
p.reshape((4,-1))
```

Out[99]:

```
array([[ 1,  2,  3],
       [ 4,  5,  6],
       [ 7,  8,  9],
       [10, 11, 12]])
```

`resize`

You can also `resize` arrays, which means to reshape **and** either truncate or pad the entries as necessary. There are two **slightly different** methods for doing this.

- `numpy.resize(A, new_shape)` returns a **new** array made by unraveling `A` and filling `new_shape` by **repeatedly cycling through A if necessary**.

In [100]:

```
print(repr(p))
np.resize(p,(2,3))
```

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

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

Out[100]:

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

In [101]:

```
np.resize(p,(3,10))
```

Out[101]:

```
array([[ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10],
       [11, 12,  1,  2,  3,  4,  5,  6,  7,  8],
       [ 9, 10, 11, 12,  1,  2,  3,  4,  5,  6]])
```

- `A.resize(new_shape)` **updates** the contents of `A` to reflect the `new_shape`. If there is not enough entries in `A`, **pads all extra with zero**.

In [102]:

```
print(repr(b))
b.resize(4,4)
print(repr(b))
```

```
array([[4],
       [7],
       [9]])
array([[4, 7, 9, 0],
       [0, 0, 0, 0],
       [0, 0, 0, 0],
       [0, 0, 0, 0]])
```

Warning. As we will mention in the next section, `numpy arrays` don't always **own** their data. `A.resize()` only works for arrays that manage their own data and aren't **views** into other arrays.

dimension shuffling

If you want to shuffle the dimensions of your array (i.e. change which axis is which) you can use **`A.transpose`** as follows

In [103]:

```
# note, you can specify the shape
# to reshape as a tuple, or arguments
a = np.arange(3*2*4).reshape(3,2,4)
print(repr(a))
```

```
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]]])
```

In [104]:

```
# tranpose using a permutation of
# of the axis indexes
b = a.transpose(1,2,0)
print(repr(b))
```

```
array([[[ 0,  8, 16],
        [ 1,  9, 17],
        [ 2, 10, 18],
        [ 3, 11, 19]],

       [[ 4, 12, 20],
        [ 5, 13, 21],
        [ 6, 14, 22],
        [ 7, 15, 23]]])
```

The short-hand for the **reversed** permutation $(0, 1, 2, \ldots, n) \mapsto (n, n-1, \ldots, 0)$ is given by `A.T` or `A.transpose()` with no arguments.

In [105]:

```
a = np.arange(2*2).reshape(2,2)
print(repr(a))
print(repr(a.T))
```

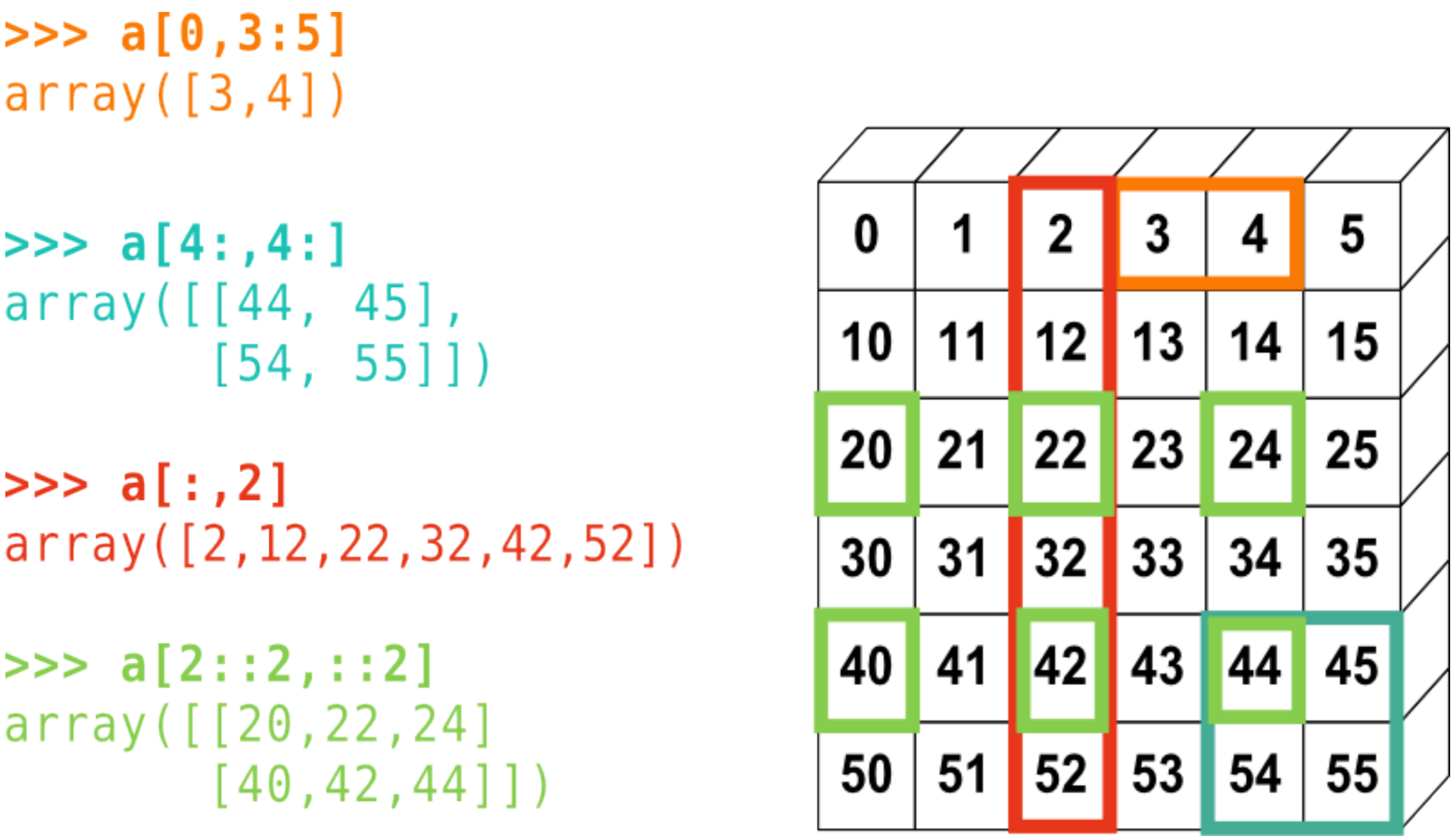
```
array([[0, 1],
       [2, 3]])
array([[0, 2],
       [1, 3]])
```

For matrices, this is the usual transpose from linear algebra.

Slicing and views

One can **slice** arrays in a very similar manner to python lists by using

`A[start_0 : end_0 : step_0, start_1 : end_1, step_1, ... ].`



For example :

In [106]:

```
a = np.arange(4*4).reshape(4,4)
print(repr(a))
# just like with lists, we don't
# need to include start, end, or step
b = a[0:2,2:]
print(repr(b))
```

```
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11],
       [12, 13, 14, 15]])
array([[2, 3],
       [6, 7]])
```

In [107]:

```
b = a[::2,::3]
print(repr(b))
```

```
array([[ 0,  3],
       [ 8, 11]])
```

Important : The arrays returned by a slice are **views** into the **data of the original**. Thus, if you **modify the data in a view you modify the original data!**

In [108]:

```
b[0,0] = 17
print(repr(b))
print(repr(a))
```

```
array([[17,  3],
       [ 8, 11]])
array([[17,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11],
       [12, 13, 14, 15]])
```

Important : A lot of other operations (such as broadcast) also share the data with the original! If you need to modify something, but you aren't sure if data is shared you can check with `numpy.may_share_memory(a,b)`

In [109]:

```
np.may_share_memory(a,b)
```

Out[109]:

```
True
```

or you can always just **make a copy!** (though you may not want to for efficiency reasons.)

In [110]:

```
c = b.copy()
np.may_share_memory(a,c)
```

Out[110]:

```
False
```

Assigning several values at once

You can use a view to assign a bunch of values at once :

In [111]:

```
print(repr(a))
b = a[::2,::3]
# assign all the same
b[:] = -75
print(repr(a))
```

```
array([[17,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11],
       [12, 13, 14, 15]])
array([[-75,   1,   2, -75],
       [  4,   5,   6,   7],
       [-75,   9,  10, -75],
       [ 12,  13,  14,  15]])
```

In [112]:

```
# assign from another array
b[:] = np.zeros((2,2))
print(repr(a))
```

```
array([[ 0,  1,  2,  0],
       [ 4,  5,  6,  7],
       [ 0,  9, 10,  0],
       [12, 13, 14, 15]])
```

In [113]:

```
# assign using broadcasting
b[:] = np.array([57,180])
print(repr(a))
```

```
array([[ 57,   1,   2, 180],
       [  4,   5,   6,   7],
       [ 57,   9,  10, 180],
       [ 12,  13,  14,  15]])
```

Fancy indexing

You can also do some fancy indexing with `numpy arrays` that goes beyond slicing. However, **fancy indexing returns copies not views!**

using tuples

We can give coordinates for each axis as tuples or lists.

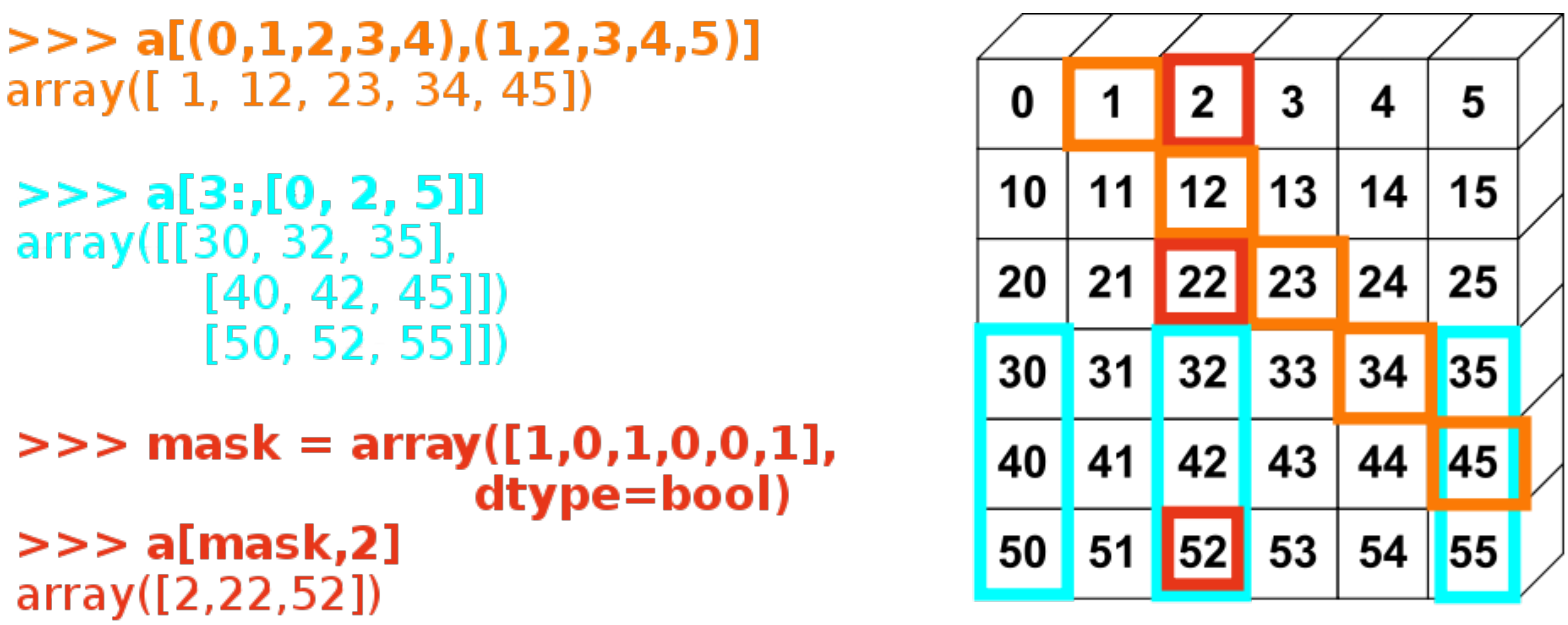
For example,

In [114]:

```
print(repr(a))
b = a[(1,3),(0,2)]
print(repr(b))
a[(1,3),(0,2)] = 8900
print(repr(a))
```

```
array([[ 57,   1,   2, 180],
       [  4,   5,   6,   7],
       [ 57,   9,  10, 180],
       [ 12,  13,  14,  15]])
array([ 4, 14])
array([[  57,    1,    2,  180],
       [8900,    5,    6,    7],
       [  57,    9,   10,  180],
       [  12,   13, 8900,   15]])
```

Here is a nice image from our text



masks or boolean arrays

You can also specify a **boolean array** as the indexes you want as you can see in the image above. Another good example is :

In [115]:

```
# this is called a boolean mask
t = (a % 3 == 2)
print(repr(t))
```

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

In [116]:

```
print(repr(a))
b = a[t]
print(repr(b))
# you can also use
# a[a % 3 == 2] = 7700
a[t] = 7700
print(repr(a))
```

```
array([[  57,    1,    2,  180],
       [8900,    5,    6,    7],
       [  57,    9,   10,  180],
       [  12,   13, 8900,   15]])
array([   2, 8900,    5, 8900])
array([[  57,    1, 7700,  180],
       [7700, 7700,    6,    7],
       [  57,    9,   10,  180],
       [  12,   13, 7700,   15]])
```