PY410 / 505
Computational Physics 1

Salvatore Rappoccio

Becoming pythonistas

- Python is a very clear language
- It is also very customizable, so you can hack very effectively
  - Many bad practices are possible to reduce clarity

- Writing CLEAR python code is “pythonic”
- People who can write pythonically are “pythonistas”

This is Jane.

Jane is a pythonista.

Be like Jane.

Becoming pythonistas

```
>>> import this
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
```

Becoming pythonistas

- Style guides:
  - http://docs.python-guide.org/en/latest/writing/style/

- Many things there, but we won't go over all explicitly

python: enumerate

- Recall from last lecture:

```
for i in [0,1,2,3] :
    print i
```

- Suppose you have some code in python to loop over objects and print them, and their index:

```
myobjs = getListFromSomewhere()
for i in xrange( len(myobjs) ) :
    print i
    print myobjs[i]
```

- Another (better) way: use “enumerate”:

```
myobjs = getListFromSomewhere()
for i,myobj in enumerate(myobjs) :
    print i
    print myobj
```

  – Typically clearer, and you don’t need the subscript everywhere

python: numpy

- “numpy” is a library written in C++ that can be used in python
  - Fast operations on lists (arrays)
  - Matrix operations
- To use in python just “import numpy”
- Nice tutorial: https://www.datacamp.com/community/tutorials/python-numpy-tutorial
- Our examples: numpy_examples.py

python : numpy

- Many ways to allocate arrays.
- Simplest case:

```
a = np.zeros(3)
print(a)
```

  - Prints “[0. 0. 0.]”

- Can also get zeros in the same dimension as another with “zeros_like”:

```
b = np.zeros_like(a)
print(b)
```

python : numpy

- Other options:
  - “ones”
  - “linspace”
  - “arange”

python : numpy

- What if you don't want to use zeros? Can use a "linspace":

Start at 0 (inclusive)

End at 2 (inclusive)

Have 41 spaces (so the 2 is the 41st space…)

```
>>> numpy.linspace(0,2,41)
array([ 0.  ,  0.05,  0.1 ,  0.15,  0.2 ,  0.25,  0.3 ,  0.35,  0.4 ,
        0.45,  0.5 ,  0.55,  0.6 ,  0.65,  0.7 ,  0.75,  0.8 ,  0.85,
        0.9 ,  0.95,  1.  ,  1.05,  1.1 ,  1.15,  1.2 ,  1.25,  1.3 ,
        1.35,  1.4 ,  1.45,  1.5 ,  1.55,  1.6 ,  1.65,  1.7 ,  1.75,
        1.8 ,  1.85,  1.9 ,  1.95,  2.  ])
```

python : numpy

- Or can use “arange”:

Start at 0 (inclusive)

End at 2 (exclusive)

Increment by 0.1

```
>>> numpy.arange(0,2, 0.1)
array([ 0. ,  0.1,  0.2,  0.3,  0.4,  0.5,  0.6,  0.7,  0.8,  0.9,  1. ,
        1.1,  1.2,  1.3,  1.4,  1.5,  1.6,  1.7,  1.8,  1.9])
```

python : numpy

- We will now be introduced to slices
- These exist in python AND in C++, but C++ only implements this in “valarrays” (similar to vector, but no iterators and math operations are faster).
- We will focus on the python slices because they are very popular

python : numpy

- Suppose you have some list:

```
>>> s = [0,1,2,3,4,5]
>>> s
[0, 1, 2, 3, 4, 5]
```

- Now you want the interior elements and exclude the exterior. What do you do?
  - Slice syntax:

```
s[begin:end:stride]
```

Begin element (inclusive)

End element (exclusive)

How many to skip

python : numpy

- Examples:

```
>>> s[1:1]
[]
>>> s[1:2]
[1]
>>> s[1:3]
[1, 2]
>>> s[:3]
[0, 1, 2]
>>> s[:3:2]
[0, 2]
```

Starts at 1, goes to 1, hence empty

Starts at 1, goes to 2 (exclusive)

Starts at 1, goes to 3 (exclusive)

Starts at beginning, goes to 3 (exclusive)

Starts at beginning, goes to 3 (exclusive), skip by 2

python : numpy

- Can also have NEGATIVE values, in which case it starts from the end:

```
>>> s[-1:]
[5]
>>> s[-2::2]
[4]
>>> s[-2::-2]
[4, 2, 0]
>>> s[-2:1:-2]
[4, 2]
```

Starts at end-1, goes to end

Starts at end-2, goes to end, skip by 2

Starts at end-2, goes to beginning, skip by -2 (i.e. backwards!)

Starts at end-2, goes to beginning +1, skip by -2 (i.e. backwards!)

python : 2d arrays

- Extending to 2d is easy, just change the 1-d length to a 2-d tuple:

  ```
  a2 = np.zeros((3,2))
  ```

  – Now you can see why “zeros_like” is useful

- To see what the shape of the array is, you just look at “shape”:

  ```
  print(a2.shape)
  ```

  – Prints (3,2)

- Can also transpose with “x.T”

python : ndarrays

- You can RESHAPE an array to be a different dimension

```
a3 = np.arange(12)
print(a3)

b3 = a3.reshape((6,2))
print(b3)

b4 = a3.reshape((3,4))
print(b4)

b5 = a3.reshape((3,2,2))
print(b5)
```

```
[ 0  1  2  3  4  5  6  7  8  9 10 11]

[[ 0  1]
 [ 2  3]
 [ 4  5]
 [ 6  7]
 [ 8  9]
 [10 11]]

[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]]

[[[ 0  1]
  [ 2  3]]

 [[ 4  5]
  [ 6  7]]

 [[ 8  9]
  [10 11]]]
```

python : ndarrays

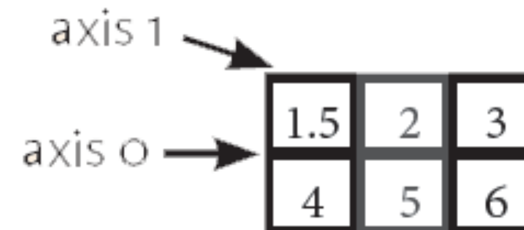
- Why does "reshape" work?

- Because, just like in C++, the arrays are ALWAYS stored as a contiguous 1d array in memory
- The interpretation is the "shape"

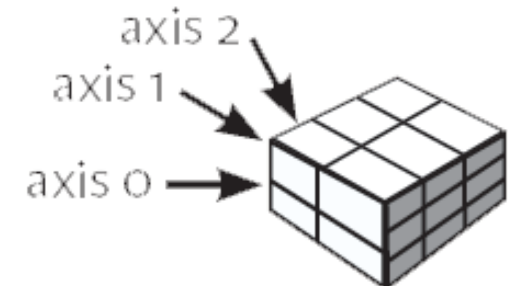
- So 12 blocks in memory can be:
  - 1d array of 12 elements
  - 2d array, 6x2
  - 2d array, 4x3
  - 3d array, 2x2x3
  - etc!

1D array

2D array



3D array



https://www.datacamp.com/community/tutorials/python-numpy-tutorial

python : Slices

- Slices also work in 2d, with a comma separating:

```
>>> for ix in xrange(5):
...     for iy in xrange(5):
...         u[ix][iy] = (ix + 1) + 100 * (iy+1)
...
>>>
>>> u
array([[ 101.,  201.,  301.,  401.,  501.],
       [ 102.,  202.,  302.,  402.,  502.],
       [ 103.,  203.,  303.,  403.,  503.],
       [ 104.,  204.,  304.,  404.,  504.],
       [ 105.,  205.,  305.,  405.,  505.]])
```

- Then can slice:

```
>>> u[1:,2:]
array([[ 302.,  402.,  502.],
       [ 303.,  403.,  503.],
       [ 304.,  404.,  504.],
       [ 305.,  405.,  505.]])
```

Careful! This graphically has x,y switched, because the “x” here is the FIRST index, and the “y” here is the SECOND index… be aware of that.

python : Slices

- Then combine to make 2d arrays with “meshgrid”:

```
>>> x= numpy.arange(0, 2, 0.5)
>>> y= numpy.arange(0, 2, 0.5)
>>> x
array([ 0. ,  0.5,  1. ,  1.5])
>>> y
array([ 0. ,  0.5,  1. ,  1.5])
>>> xv,yv = numpy.meshgrid(x,y)
>>> xv
array([[ 0. ,  0.5,  1. ,  1.5],
       [ 0. ,  0.5,  1. ,  1.5],
       [ 0. ,  0.5,  1. ,  1.5],
       [ 0. ,  0.5,  1. ,  1.5]])
>>> yv
array([[ 0. ,  0. ,  0. ,  0. ],
       [ 0.5,  0.5,  0.5,  0.5],
       [ 1. ,  1. ,  1. ,  1. ],
       [ 1.5,  1.5,  1.5,  1.5]])
```

Create vectors x and y

Print x and y

Create the meshgrid

xv holds copies of [0,0.5,1,1.5] in increasing x

yv holds copies of [0,0.5,1,1.5] in increasing y

python : Slices

- Can also be left-hand-slides of assignment!

```
>>> u[1:] = 56784
>>> u
array([[   101.,    201.,    301.,    401.,    501.],
       [ 56784.,  56784.,  56784.,  56784.,  56784.],
       [ 56784.,  56784.,  56784.,  56784.,  56784.],
       [ 56784.,  56784.,  56784.,  56784.,  56784.],
       [ 56784.,  56784.,  56784.,  56784.,  56784.]])
```

python : operations on lists

- Lots of neat little operations you can use
- Builtin functions:
  - https://docs.python.org/3/library/functions.html

- Examples:
  - sum(a) : computes sum of elements of a
  - any(a): return true if any elements are true
  - min(a): returns minimum element of a
  - sorted(a): returns a, but sorted

- To be a pythonista, best to check if there are pythonic tools for your use case!

python : numpy operations

- Operations on numpy arrays are very powerful
- We’re going to come back to this later.

python : advanced use of class members

- In C++: need to know ahead of time which function you're calling, which member you're accessing, etc.
- In python, no such restriction:
  - getattr(a, "val"): returns a.val
  - setattr(a, "val", b): performs a.val = b

  - Really useful! "val" can be a variable input by the user!

Matplotlib

- Lots of graphics tools used in python
- Matplotlib is one industry standard
- Tutorial here:
  - https://matplotlib.org/users/pyplot_tutorial.html

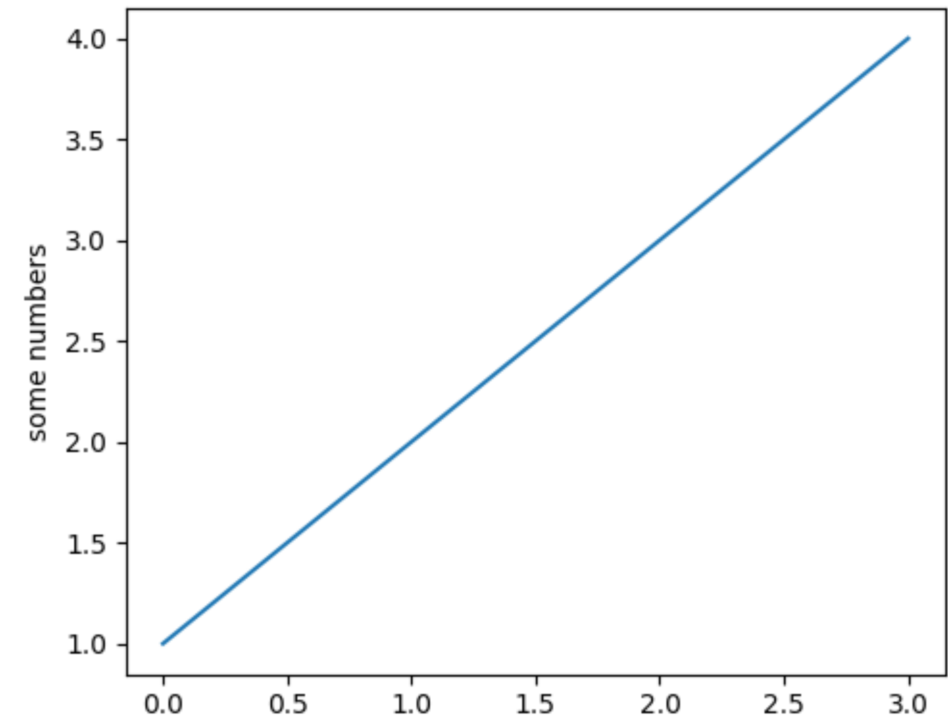
Import code

Tell it what to plot

Label the y axis

Draw the actual plot itself

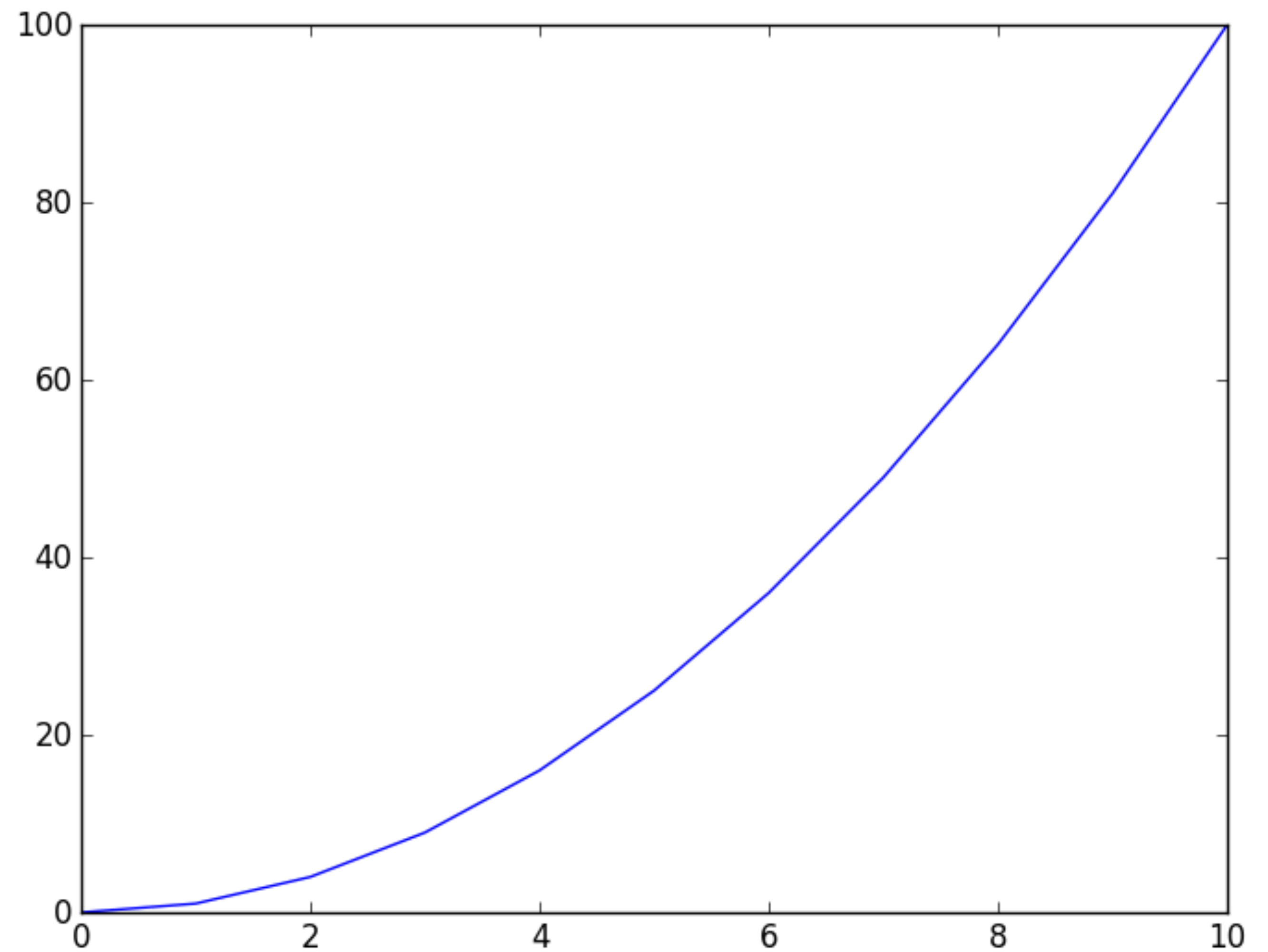
```
import matplotlib.pyplot as plt
plt.plot([1,2,3,4])
plt.ylabel('some numbers')
plt.show()
```



Matplotlib

- Example: “Matplotlib” examples

```
import matplotlib.pyplot as plt
from read_plot import read_plot
[x,y] = read_plot("histogram.dat")
plt.plot(x,y)
plt.show()
```



Matplotlib

- I will be using this throughout the semester, and so will you
- If you want to learn a bit about this, the tutorials are good and easy to use. I recommend them!