UC Berkeley Extension

Python for Data Analysis and Scientific Computing

X433.3 (2 semester units in COMPSCI)

Instructor Alexander I. Iliev, Ph.D.

Course Content Outline

- **NumPy 2/3**
- Array operations
- Reductions
- Broadcasting
- Array: shaping, reshaping, flattening, resizing, dimension changing
- Data sorting
- Good coding practices

- **NumPy 3/3**
- Type casting
- Masking data
- Organizing arrays
- Loading data files
- Dealing with polynomials Midterm, Project proposal due

- **Scipy 1/2**
- What is Scipy?
- Working with files
- Algebraic operations
- The Fast Fourier Transform
- Signal Processing

- **Scipy 2/2**
- Interpolation
- Statistics
- Optimization
- Multithreading and Multiprocessing

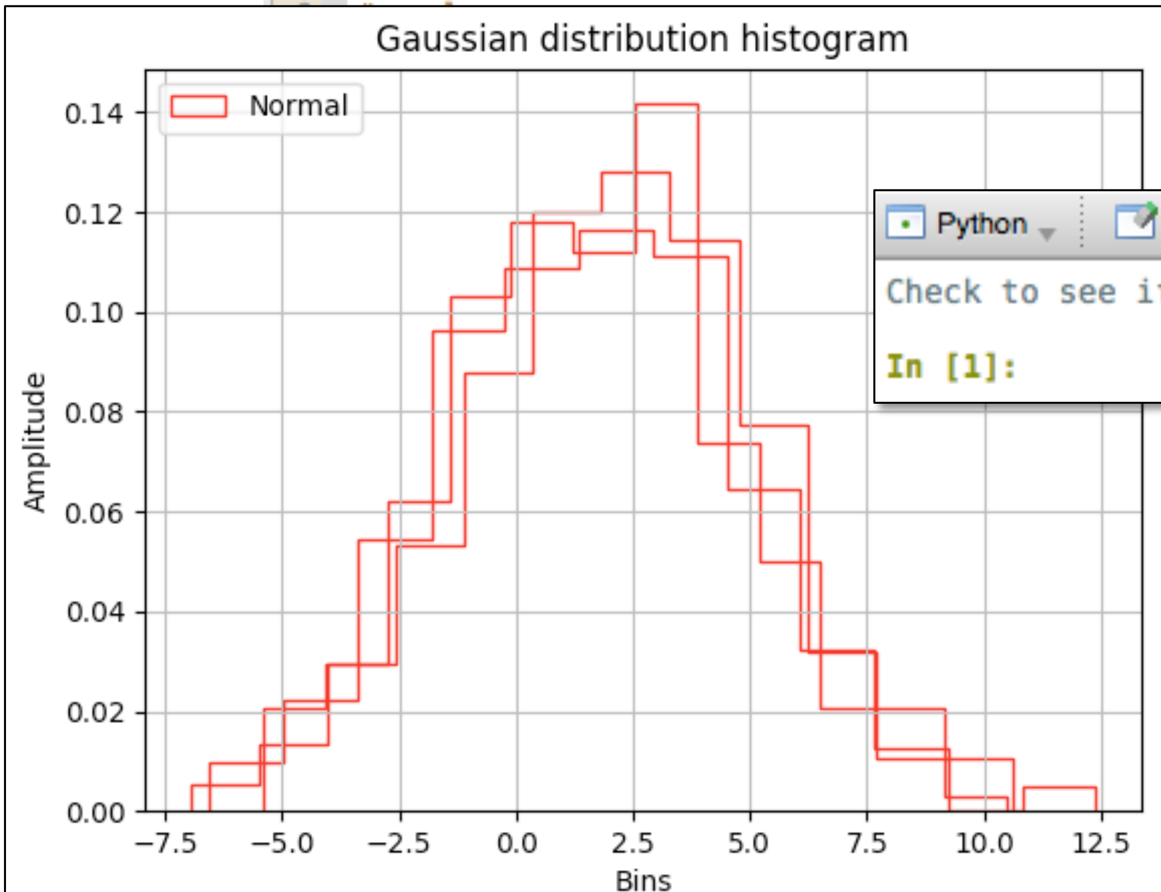
- **Project**
- Project Presentation Final Project

Solution

```python
## Class exercise in Lecture 5:
# 1. Use random from numpy:
import pylab as plb

# 2. Create a dictionary with 5 keys and empty valies in A:
A = {'home': '', 'office': '', 'bar': '', 'hospital': '', 'library': ''}

# 3. Using a function, assign random values to each key between [0:10], using a for
#    loop ... :
def my_function_1(x):
    for index, key in enumerate(x):
        x[key] = plb.randint(0,11)
    return(x)
# ... and return the result in B:
B = my_function_1(A)

# 4. Using a function, change the value of any member in B that is less than 5
#    with the result from (4.1) (consider using an if statement in a loop):
def my_function_2(y):
    for index, key in enumerate(y):
        if y[key] < 5:
            # 4.1 Using normal distrib. with mean=2 and std=3 create a 256pts array:
            y[key] = plb.normal(2,3,size=256)
            # 4.2 Using a histogram with 12 bins, plot the result from 4.1:
            plb.figure('Histogram'), plb.title("Gaussian distribution histogram")
            plb.hist(y[key], bins=12, histtype='step', normed=True, \
            color='red', label='Normal'), plb.legend(loc='upper left')
            plb.xlabel("Bins"), plb.ylabel("Amplitude"), plb.grid(True), plb.pause(1)
    return(y)
# 5. Assign the result from 4. in C:
C = my_function_2(B)

# 6. Update one of the keys in C with another using the ‘pop’ feature:
A['store'] = A.pop('bar') # The value of the old key is assigned to the new key

# 7. Update another key in C manually (add the new one and delete the old one):
A['shop']=A['library']
del A['library']

# 8. Compare them using a short conditional expression:
print('Check to see if A, B and C are equal: ', A is B is C)
```

Solution

```python
## Class exercise in Lecture 5:
# 1. Use random from numpy:
import pylab as plb

# 2. Create a dictionary with 5 keys and empty valies in A:
A = {'home': '', 'office': '', 'bar': '', 'hospital': '', 'library': ''}

# 3. Using a function, assign random values to each key between [0:10], using a for
```




```
Python
Check to see if A, B and C are equal:  True

In [1]:
```

```python
an=2 and std=3 create a 256pts array:

s, plot the result from 4.1:
"Gaussian distribution histogram")
'step', normed=True, \
egend(loc='upper left')
litude"), plb.grid(True), plb.pause(1)

sing the ‘pop’ feature:
d key is assigned to the new key

# 7. Update another key in C manually (add the new one and delete the old one):
A['shop']=A['library']
del A['library']

# 8. Compare them using a short conditional expression:
print('Check to see if A, B and C are equal: ', A is B is C)
```

Array operations

- Arrays operations

```
import numpy as np
        x = np.array([[1,2],[3,4]], dtype=np.float64)
        y = np.array([[5,6],[7,8]], dtype=np.float64)

# Elementwise sum; both produce the array # [[ 6.0 8.0], [10.0 12.0]]
print(x + y)
print(np.add(x, y))

# Elementwise difference; both produce the array # [[-4.0 -4.0], [-4.0 -4.0]]
print(x - y)
print(np.subtract(x, y))

# Elementwise product; both produce the array # [[ 5.0 12.0], [21.0 32.0]]
print(x * y)
print(np.multiply(x, y))

# Elementwise division; both produce the array # [[ 0.2 0.33333333], [ 0.42857143 0.5 ]]
print(x / y)
print(np.divide(x, y))

# Elementwise square root; produces the array # [[ 1. 1.41421356], [ 1.73205081 2. ]]
print(np.sqrt(x))
```

Array operations

- Array operations
  - basic operations of arrays with scalars are very simple and intuitive
  - they performed on element by element basis

Examples:

```
In [1]: import pylab as plb

In [2]: a = plb.array([12,34,41,54,68,72])

In [3]: a
Out[3]: array([12, 34, 41, 54, 68, 72])

In [4]: a=a+5

In [5]: a
Out[5]: array([17, 39, 46, 59, 73, 77])

In [6]: a=a*2

In [7]: a
Out[7]: array([ 34,  78,  92, 118, 146, 154])

In [8]: a=a-1

In [9]: a
Out[9]: array([ 33,  77,  91, 117, 145, 153])
```

Array operations

- Array operations
  - As in any language we need to be aware of NumPy’s precedence:
    - ( ) – has the highest precedence order. The inner most bracket expression has the highest precedence
    - F (args...) – Function calls
    - x[ind:ind] – Slicing
    - x[index] – Subscription
    - x.attribute – Attribute reference
    - ** – power, exponentiation
    - ~x – bitwise not
    - +x, -x – positive and negative signs
    - *, /, % – multiplication, division, remainder
    - +, - – addition, subtraction
    - <<, >> – bitwise shift
    - & – bitwise and
    - ^ – bitwise xor (exclusive *or*, meaning both values must be mutually exclusive)
    - | – bitwise or
    - in, is, not in, is not, <, <=, >, >=, <>, !=, == – comparison, membership and identity tests
    - not x – boolean NOT
    - and – boolean AND
    - or – boolean OR

conversions: bin(x) – int to bin, int(‘0011001’, 2) – bin to int

Examples:

e & f | z == (e & f) | z   e | f & z == e | (f & z) ,   where z = e & f

Array operations

- Array operations
  - associativity:
    - (12 – 32) + 41 = 21 → left-associative
    - 12 – (32 + 41) = - 61 → right-associative

Examples:

```
In [10]: a**1/2
Out[10]: array([ 16.5,  38.5,  45.5,  58.5,  72.5,  76.5])

In [11]: (a**1)/2
Out[11]: array([ 16.5,  38.5,  45.5,  58.5,  72.5,  76.5])

In [12]: a**(1/2)
Out[12]:
array([  5.74456265,   8.77496439,   9.53939201,  10.81665383,
        12.04159458,  12.36931688])

In [13]: plb.sqrt(a)
Out[13]:
array([  5.74456265,   8.77496439,   9.53939201,  10.81665383,
        12.04159458,  12.36931688])
```

Array operations

- Array operations
  - arithmetic operations between arrays are also performed element by element
  - NumPy operations are much faster than the ones used by basic math in Python

Examples:

```
In [14]: b = plb.arange(6)+4

In [15]: b
Out[15]: array([4, 5, 6, 7, 8, 9])

In [16]: a-b
Out[16]: array([ 29,  72,  85, 110, 137, 144])

In [17]: a/b
Out[17]:
array([  8.25      ,  15.4       ,  15.16666667,  16.71428571,
        18.125     ,  17.        ])

In [18]: c = plb.ones(6)

In [19]: c=(a+b)**2-plb.sqrt(c)

In [20]: c
Out[20]: array([  1368.,   6723.,   9408.,  15375.,  23408.,  26243.])

In [21]: d = plb.arange(5)

In [22]: plb.size(a)
Out[22]: 6

In [23]: plb.size(d)
Out[23]: 5

In [24]: a-d
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-24-ba0eb01b3f36> in <module>()
----> 1 a-d

ValueError: operands could not be broadcast together with shapes (6,) (5,)
```

when performing operations between arrays they must -> be of the same shape

Array operations

- Array operations
  - logical operations are built in NumPy

Examples:

```
In [25]: e = plb.array([0, 1, 1, 0], dtype=bool)

In [26]: f = plb.array([1, 0, 1, 0], dtype=bool)

In [27]: e
Out[27]: array([False,  True,  True, False], dtype=bool)

In [28]: f
Out[28]: array([ True, False,  True, False], dtype=bool)

In [29]: plb.logical_and(e,f)
Out[29]: array([False, False,  True, False], dtype=bool)

In [30]: plb.logical_or(e,f)
Out[30]: array([ True,  True,  True, False], dtype=bool)

In [31]: plb.logical_xor(e,f)
Out[31]: array([ True,  True, False, False], dtype=bool)

In [32]: plb.logical_not(e)
Out[32]: array([ True, False, False,  True], dtype=bool)

In [33]: plb.logical_not(f)
Out[33]: array([False,  True, False,  True], dtype=bool)
```

Array operations

- Array operations
  - arrays in NumPy can be compared in a element by element basis easily:

Examples:

```
In [34]: e
Out[34]: array([False,  True,  True, False], dtype=bool)

In [35]: f
Out[35]: array([ True, False,  True, False], dtype=bool)

In [36]: e == f
Out[36]: array([False, False,  True,  True], dtype=bool)

In [37]: e > f
Out[37]: array([False,  True, False, False], dtype=bool)

In [38]: e < f
Out[38]: array([ True, False, False, False], dtype=bool)
```

Array operations

- Array operations
  - arrays in NumPy can also be compared as whole vectors:

Examples:

```
In [39]: g = e

In [40]: h = plb.array([0, 1, 1, 0], dtype=bool)

In [41]: e
Out[41]: array([False,  True,  True, False], dtype=bool)

In [42]: f
Out[42]: array([ True, False,  True, False], dtype=bool)

In [43]: g
Out[43]: array([False,  True,  True, False], dtype=bool)

In [44]: h
Out[44]: array([False,  True,  True, False], dtype=bool)

In [45]: plb.array_equal(e,f)
Out[45]: False

In [46]: plb.array_equal(e,g)
Out[46]: True

In [47]: plb.array_equiv(e,g)
Out[47]: True

In [48]: plb.array_equiv(g,h)
Out[48]: True

In [49]: id(e)
Out[49]: 4543434224

In [50]: id(g)
Out[50]: 4543434224

In [51]: id(h)
Out[51]: 4582111072
```

Array operations

- Array operations
  - one can take the *sin*, *cos*, *tan*, *log*, *exp* or *exp2* of an array easily in NumPy:

Examples:

be aware of your data to avoid this ->

```
In [52]: i = plb.arange(5)-2

In [53]: i
Out[53]: array([-2, -1,  0,  1,  2])

In [54]: plb.sin(i)
Out[54]: array([-0.90929743, -0.84147098,  0.        ,  0.84147098,  0.90929743])

In [55]: plb.cos(i)
Out[55]: array([-0.41614684,  0.54030231,  1.        ,  0.54030231, -0.41614684])

In [56]: plb.tan(i)
Out[56]: array([ 2.18503986, -1.55740772,  0.        ,  1.55740772, -2.18503986])

In [57]: plb.log(i)
__main__:1: RuntimeWarning: divide by zero encountered in log
__main__:1: RuntimeWarning: invalid value encountered in log
Out[57]: array([        nan,         nan,        -inf,  0.        ,  0.69314718])

In [58]: plb.exp(i)
Out[58]: array([ 0.13533528,  0.36787944,  1.        ,  2.71828183,  7.3890561 ])

In [59]: plb.exp2(i)
Out[59]: array([ 0.25,  0.5 ,  1.  ,  2.  ,  4.  ])
```

Array operations

- Array operations
  - Transpose an array is in NumPy is performed like this:

Examples:

```
In [60]: j = plb.tril(plb.ones((5,5)),-2) # Lower triangle of an array

In [61]: j
Out[61]:
array([[ 0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.],
       [ 1.,  0.,  0.,  0.,  0.],
       [ 1.,  1.,  0.,  0.,  0.],
       [ 1.,  1.,  1.,  0.,  0.]])

In [62]: j.T
Out[62]:
array([[ 0.,  0.,  1.,  1.,  1.],
       [ 0.,  0.,  0.,  1.,  1.],
       [ 0.,  0.,  0.,  0.,  1.],
       [ 0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.]])

In [63]: k = plb.triu(plb.ones((5,5)),2) # Upper triangle of an array

In [64]: k
Out[64]:
array([[ 0.,  0.,  1.,  1.,  1.],
       [ 0.,  0.,  0.,  1.,  1.],
       [ 0.,  0.,  0.,  0.,  1.],
       [ 0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.]])

In [65]: plb.array_equal(j.T,k)
Out[65]: True
```

Reductions

- Reductions
  - reduction is an array operation in which a special set of array elements is produced based on a selection or mathematical operation
  - In NumPy there are several methods that process arrays as input and can create a new set of array elements as output based on specific requirements
  - one of the most basic methods for reduction is *sum*

Examples:

Python

```
In [66]: l = plb.arange(12).reshape(2,3,2)

In [67]: l
Out[67]:
array([[[ 0,  1],
        [ 2,  3],
        [ 4,  5]],

       [[ 6,  7],
        [ 8,  9],
        [10, 11]]])

In [68]: m = plb.sum(l)

In [69]: m
Out[69]: 66
```

Reductions

- Reductions
  - *sum* can be performed on specific set of elements from an array such as rows and columns:

Examples:

```
In [70]: n = plb.array([[12,34,41],[52,64,72]])

In [71]: n
Out[71]:
array([[12, 34, 41],
       [52, 64, 72]])

In [72]: n.sum()
Out[72]: 275

In [73]: n[:,:].sum()    # show total sum of all elements in 'n' (as above)
Out[73]: 275

In [74]: n[:,2].sum()    # shows the sum of column #3
Out[74]: 113

In [75]: n[1,:].sum()    # shows the sum of row #2
Out[75]: 188

In [76]: n.sum(axis=0)   # shows the sum of each column
Out[76]: array([ 64,  98, 113])

In [77]: n.sum(axis=1)   # shows the sum of each row
Out[77]: array([ 87, 188])

In [78]: n.cumsum()      # shows the cumulative sum in 'n'
Out[78]: array([ 12,  46,  87, 139, 203, 275])
```

Reductions

- Reductions
  - logical operations can be performed on rows and columns:

Examples:

```
In [79]: n
Out[79]:
array([[12, 34, 41],
       [52, 64, 72]])

In [80]: p = plb.array([[12,34,41,0],[52,64,0,72]])

In [81]: p
Out[81]:
array([[12, 34, 41,  0],
       [52, 64,  0, 72]])

In [82]: plb.any(p != 0)     # show if there are any elements different than '0'
Out[82]: True

In [83]: plb.all(p != 0)     # show if all elements are different than '0'
Out[83]: False

In [84]: plb.any(p != n)     # show if any elements in 'p' and 'n' are different
Out[84]: True

In [85]: n!=p                # show if 'n' is different than 'p'
Out[85]: True

In [86]: n.sum()!=p.sum()    # show if total sum in 'n' is different than the one in 'p'
Out[86]: False

In [87]: n.sum()==p.sum()    # show if total sum in 'n' and 'p' is the same
Out[87]: True
```

Reductions

- Reductions
  - *min, max* extremes can be performed on specific set of elements from an array such as rows and columns:

Examples:

```
In [88]: n
Out[88]:
array([[12, 34, 41],
       [52, 64, 72]])

In [89]: n.min()        # shows the min value in the entire array
Out[89]: 12

In [90]: n.argmin()     # shows the index of the min value
Out[90]: 0

In [91]: n.max()        # shows the max value in the entire array
Out[91]: 72

In [92]: n.argmax()     # shows the index of the max value
Out[92]: 5

In [93]: n.max(0)       # shows the max value column
Out[93]: array([52, 64, 72])

In [94]: n.max(axis=0)  # same as above
Out[94]: array([52, 64, 72])

In [95]: n.max(1)       # shows the max value row
Out[95]: array([41, 72])

In [96]: n.max(axis=1)  # same as above
Out[96]: array([41, 72])
```

Reductions

- Reductions
  - statistics can be performed on the arrays as well as on rows and columns:

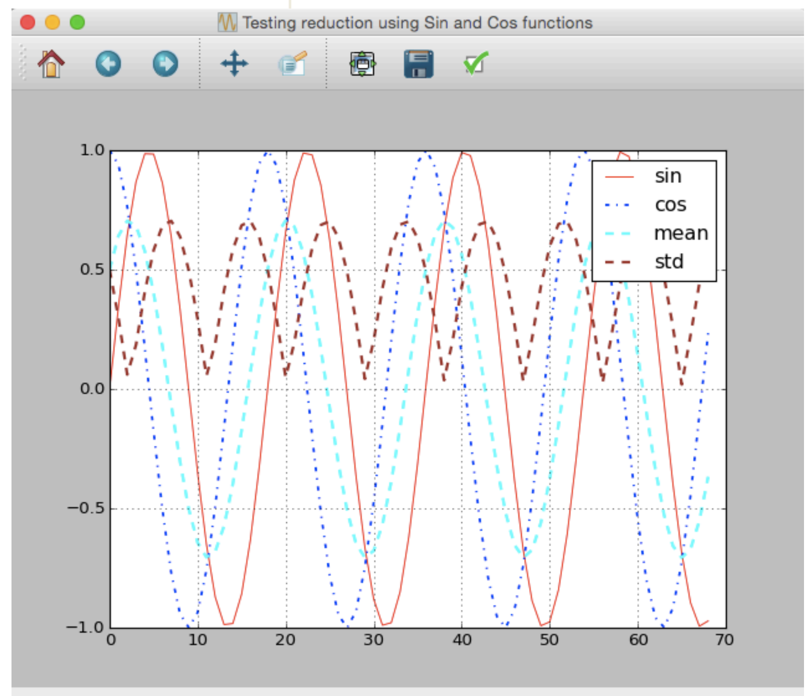
Examples:

```
In [97]: n
Out[97]:
array([[12, 34, 41],
       [52, 64, 72]])

In [98]: n.mean()             # find the mean value in 'n'
Out[98]: 45.833333333333336

In [99]: n.std()              # find the standard deviation in 'n'
Out[99]: 19.836134256004172

In [100]: n.std(axis=1)       # find the std of each row in 'n'
Out[100]: array([ 12.35583533,   8.21921867])

In [101]: plb.median(n)           # find the median in 'n'
Out[101]: 46.5

In [102]: plb.median(n,axis=0)    # find the median of each column in 'n'
Out[102]: array([ 32. ,  49. ,  56.5])
```

Reductions

- Reductions
  - taking the mean and std values of *sin* vs *cos* functions:

```python
# Example using reduction:
import pylab as plb

x0 = plb.arange(0, 24, 0.35)
x1 = plb.sin(x0)
x2 = plb.cos(x0)
fig = plb.mean([x1,x2],axis=0)
plb.figure('Testing reduction using Sin and Cos functions')
plb.plot(x1,color='r', linewidth=0.8, label='sin')
plb.plot(x2,color='b', linewidth=1.5, linestyle='-.',
                    label='cos')

# plot the mean values of the sin and cos functions:
plb.plot(plb.mean([x1,x2],axis=0), color='cyan', linewidth=2,
                    linestyle='--', label='mean')

# plot the std values of the sin and cos functions:
plb.plot(plb.std([x1,x2],axis=0), color='brown', linewidth=2,
                    linestyle='--', label='std')
plb.legend(loc='upper right')
plb.grid(True)
plb.show()
```



Broadcasting

- Broadcasting
  - normally any operation on arrays will require them to be of the same size and shape, like in Matlab
  - in some cases where this rule is not met, NumPy has a way of overcoming that problem whenever possible
  - this is called Broadcasting
  - in a simple multiplication of arrays with scalars (which we have seen so far) this is exactly what NumPy is doing
  - in order to solve the problem NymPy automatically changes the shape of some arrays or scalars by repeating some values in order to complete the task

Broadcasting

- Broadcasting
  - in this example we multiply an array with a scalar and the latter changes its shape by repeating the same scalar to match the shape of the array
  - only after this rule is satisfied the multiplication can be performed

Example:

```
In [156]: # Broacasting - multiplication example:

In [157]: r = plb.array([20,40,60,80,100])

In [158]: r
Out[158]: array([ 20,  40,  60,  80, 100])

In [159]: plb.shape(r)
Out[159]: (5,)

In [160]: s = 5

In [161]: s = r*s

In [162]: s
Out[162]: array([100, 200, 300, 400, 500])

In [163]: plb.shape(s)
Out[163]: (5,)
```

r						*	s	s - repeated					=	s				
20	40	60	80	100		*	5	5	5	5	5		=	100	200	300	400	500

Broadcasting

- Broadcasting
  - in this example we add two arrays of different sizes and shapes
  - we notice that in order for the addition to be performed both shapes were changed

Example:

```
In [164]: # Broacasting - addition example:

In [165]: r = plb.array([20,40,60,80,100]).reshape(5,1)

In [166]: r
Out[166]:
array([[ 20],
       [ 40],
       [ 60],
       [ 80],
       [100]])

In [167]: plb.shape(r)
Out[167]: (5, 1)

In [168]: s = plb.arange(2,4)

In [169]: s
Out[169]: array([2, 3])

In [170]: plb.shape(s)
Out[170]: (2,)

In [171]: t = r+s

In [172]: t
Out[172]:
array([[ 22,  23],
       [ 42,  43],
       [ 62,  63],
       [ 82,  83],
       [102, 103]])

In [173]: plb.shape(t)
Out[173]: (5, 2)
```

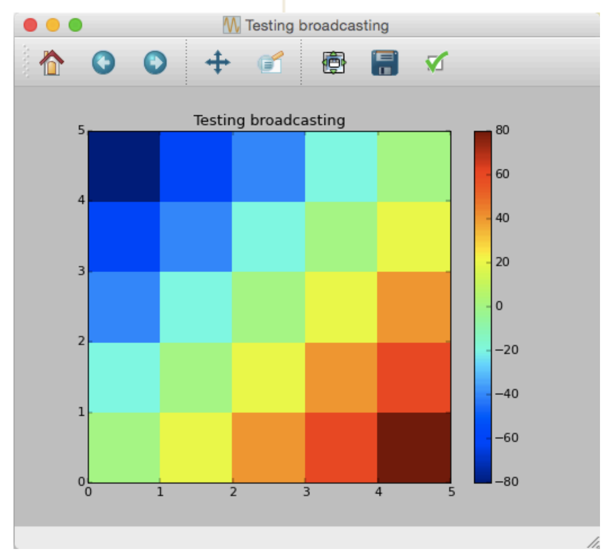
r	r - rep.						t	
20	20		2	3	s		22	23
40	40		2	3	s - repeated		42	43
60	60	+	2	3		=	62	63
80	80		2	3			82	83
100	100		2	3			102	103

Broadcasting

- Broadcasting

Example:

```
## Broadcasting - example of a 2-D array or square matrix with diagonal = 0:
r = plb.array([20,40,60,80,100])
print(r)
print(r[:,plb.newaxis].shape)   # no change is introduced to 'r'
print(r[plb.newaxis,:].shape)   # change the shape of 'r' and add a new axis - 2-D array
s = r - r[:,plb.newaxis]        # create a square matrix with diagonal = 0
print(s)

plb.figure('Testing broadcasting', dpi=65)
plb.gca(title='Testing broadcasting')
plb.pcolormesh(s)   # chosing the printing scheme
plb.colorbar()      # adding a colorbar on the side for clarity
plb.pause(1)
```



newaxis – expands the dimensions by one unit-length dimension

```
In [1]: print(r[:,plb.newaxis].shape)
(5, 1)

In [2]: print(r[plb.newaxis,:].shape)
(1, 5)
```

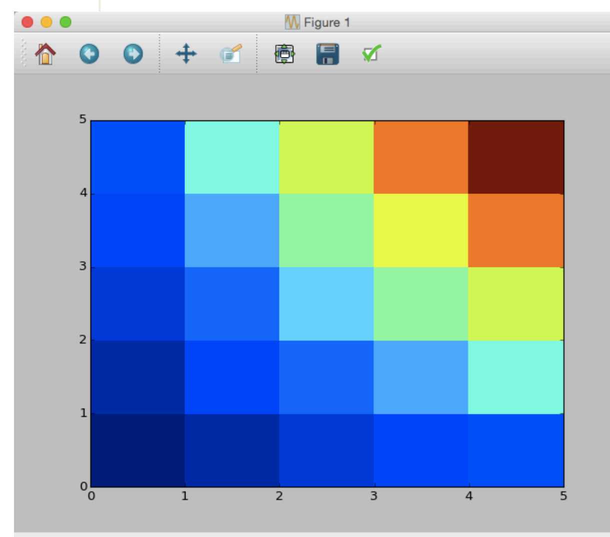
Broadcasting

- Broadcasting
  - the NymPy function *ogrid* assists us in making horizontal and vertical shape vector arrays in one line
  - we can also assign the x and y axis to different variables

Example:

```
In [174]: # Broadcasting - 'ogrid' example:

In [175]: plb.ogrid[1:6, 1:6]
Out[175]:
[array([[1],
        [2],
        [3],
        [4],
        [5]]), array([[1, 2, 3, 4, 5]])]

In [176]: u, v = plb.ogrid[1:6, 1:6]    # assign x,y axis to u,v variables

In [177]: u
Out[177]:
array([[1],
       [2],
       [3],
       [4],
       [5]])

In [178]: v
Out[178]: array([[1, 2, 3, 4, 5]])

In [179]: u*v
Out[179]:
array([[ 1,  2,  3,  4,  5],
       [ 2,  4,  6,  8, 10],
       [ 3,  6,  9, 12, 15],
       [ 4,  8, 12, 16, 20],
       [ 5, 10, 15, 20, 25]])

In [180]: plb.pcolormesh(u*v)
Out[180]: <matplotlib.collections.QuadMesh at 0x1059bc208>
```

```
(<module>)>>> u.size
5

(<module>)>>> v.size
5
```



Broadcasting

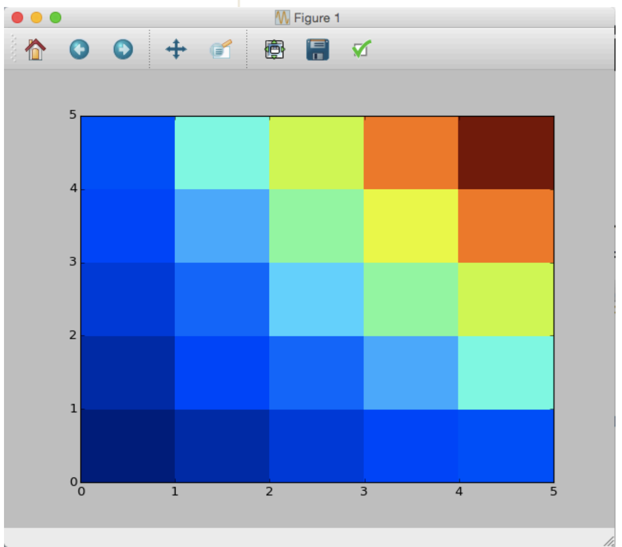
- Broadcasting
  - NymPy also provides the function *mgrid*, designed specifically to provide matrices full of indices in case we can't take a full advantage from broadcasting

Example:

```
In [181]: # Broadcasting - 'mgrid' example:

In [182]: plb.mgrid[1:6, 1:6]
Out[182]:
array([[[1, 1, 1, 1, 1],
        [2, 2, 2, 2, 2],
        [3, 3, 3, 3, 3],
        [4, 4, 4, 4, 4],
        [5, 5, 5, 5, 5]],

       [[1, 2, 3, 4, 5],
        [1, 2, 3, 4, 5],
        [1, 2, 3, 4, 5],
        [1, 2, 3, 4, 5],
        [1, 2, 3, 4, 5]]])

In [183]: u, v = plb.mgrid[1:6, 1:6]

In [184]: u*v
Out[184]:
array([[ 1,  2,  3,  4,  5],
       [ 2,  4,  6,  8, 10],
       [ 3,  6,  9, 12, 15],
       [ 4,  8, 12, 16, 20],
       [ 5, 10, 15, 20, 25]])

In [185]: plb.pcolormesh(u*v)
Out[185]: <matplotlib.collections.QuadMesh at 0x105944780>
```

```
(<module>)>>> u.size
25

(<module>)>>> v.size
25
```

we get the same result



Arrays

- Array: shaping, reshaping, flattening, resizing, dimension changing
  - array manipulation in NumPy is supported by several built in functions such as:
    shape, reshape, ravel, resize, newaxis
  - there are several common techniques we will consider in the next few slides

Using shape/reshape:

(we have already used them)

```
In [186]: w = plb.array([12,32,41,52,68,72])

In [187]: w
Out[187]: array([12, 32, 41, 52, 68, 72])

In [188]: plb.shape(w)
Out[188]: (6,)

In [189]: w = w.reshape(3,2)

In [190]: plb.shape(w)
Out[190]: (3, 2)

In [191]: w
Out[191]:
array([[12, 32],
       [41, 52],
       [68, 72]])
```

Arrays

- Array: shaping, reshaping, flattening, resizing, dimension changing
  - array manipulation in NumPy is supported by several built in functions such as:
    shape, reshape, ravel, resize, newaxis
  - there are several common techniques we will consider in the next few slides

Using ravel for flattening, hence representing a multidimensional array in a flat 1-D vector:

```
In [191]: w
Out[191]:
array([[12, 32],
       [41, 52],
       [68, 72]])

In [192]: w.ravel()
Out[192]: array([12, 32, 41, 52, 68, 72])

In [193]: w.T   # transpose
Out[193]:
array([[12, 41, 68],
       [32, 52, 72]])

In [194]: w.T.ravel()   # transpose
Out[194]: array([12, 41, 68, 32, 52, 72])
```

Arrays

- Array: shaping, reshaping, flattening, resizing, dimension changing
  - array manipulation in NumPy is supported by several built in functions such as:

    shape, reshape, ravel, resize, newaxis
  - there are several common techniques we will consider in the next few slides

Using resize for resizing for existing array:

```
In [195]: x = plb.arange(5,11,1)

In [196]: x
Out[196]: array([ 5,  6,  7,  8,  9, 10])

In [197]: x.resize(2,3)

In [198]: x
Out[198]:
array([[ 5,  6,  7],
       [ 8,  9, 10]])

In [199]: x.resize(8)
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-199-935ffac0ea7b> in <module>()
----> 1 x.resize(8)

ValueError: cannot resize an array that references or is referenced
by another array in this way.  Use the resize function

In [200]: x.resize(6)

In [201]: x
Out[201]: array([ 5,  6,  7,  8,  9, 10])
```

Arrays

- Array: shaping, reshaping, flattening, resizing, dimension changing
  - array manipulation in NumPy is supported by several built in functions such as:
    shape, reshape, ravel, resize, newaxis
  - there are several common techniques we will consider in the next few slides

Using newaxis for adding an extra dimension on an existing array:

```
Python
In [202]: y = plb.arange(10,13,0.5)

In [203]: y
Out[203]: array([ 10. ,  10.5,  11. ,  11.5,  12. ,  12.5])

In [204]: plb.shape(y)
Out[204]: (6,)

In [205]: y[:, plb.newaxis]   # ading an extra dimension with 'newaxis'
Out[205]:
array([[ 10. ],
       [ 10.5],
       [ 11. ],
       [ 11.5],
       [ 12. ],
       [ 12.5]])

In [206]: y[plb.newaxis, :]
Out[206]: array([[ 10. ,  10.5,  11. ,  11.5,  12. ,  12.5]])

In [207]: plb.shape(y[:,plb.newaxis])
Out[207]: (6, 1)

In [208]: plb.shape(y[plb.newaxis,:])
Out[208]: (1, 6)
```

Arrays

- Array: shaping, reshaping, flattening, resizing, dimension changing
  - array manipulation in NumPy is supported by several built in functions such as:

    shape, reshape, ravel, resize, newaxis
  - there are several common techniques we will consider in the next few slides

Changing and shifting dimensions on an existing array:

```
In [209]: z = plb.arange(12,15,0.5)

In [210]: z
Out[210]: array([ 12. ,  12.5,  13. ,  13.5,  14. ,  14.5])

In [211]: plb.shape(z)
Out[211]: (6,)

In [212]: z = z.reshape(2,3)

In [213]: z
Out[213]:
array([[ 12. ,  12.5,  13. ],
       [ 13.5,  14. ,  14.5]])

In [214]: z[1,2]
Out[214]: 14.5

In [215]: z = z.T

In [216]: z
Out[216]:
array([[ 12. ,  13.5],
       [ 12.5,  14. ],
       [ 13. ,  14.5]])

In [217]: z[2,1]
Out[217]: 14.5
```

z.resize assigns the change directly to z

```
In [1]: z
Out[1]:
array([[12. , 13.5],
       [12.5, 14. ],
       [13. , 14.5]])

In [2]: z.resize(6,1)

In [3]: z
Out[3]:
array([[12. ],
       [12.5],
       [13. ],
       [13.5],
       [14. ],
       [14.5]])
```

Data sorting

- Data sorting
  - array data can be sorted on rows and columns basis

Example:

```
In [218]: a = plb.rand(12)*10 # create a matrix with 12 random

In [219]:                     # elements bound between 0 and 10

In [220]: a = plb.round_(a).reshape(4,3)  # round all values and reshape the array

In [221]: a = a.astype(int)   # type cast all elements from 'float64' to 'int'

In [222]: a
Out[222]:
array([[ 3,  0,  6],
       [ 4,  6,  4],
       [10,  1,  9],
       [ 1,  6,  9]])

In [223]: a.sort(axis=0)  # sort each column (same as 'a.sort(0)')

In [224]: a
Out[224]:
array([[ 1,  0,  4],
       [ 3,  1,  6],
       [ 4,  6,  9],
       [10,  6,  9]])

In [225]: a.sort(1)       # sort each row (same as 'a.sort()' and 'a.sort(axis=1)')

In [226]: a
Out[226]:
array([[ 0,  1,  4],
       [ 1,  3,  6],
       [ 4,  6,  9],
       [ 6,  9, 10]])
```

Data sorting

- Data sorting
  - sorting indices can be returned without sorting the array by calling the argsort function

Example:

```
In [227]: a = plb.rand(12)*10 # create a matrix with 12 random

In [228]:                     # elements bound between 0 and 10

In [229]: a = plb.round_(a).reshape(4,3)  # round all values and reshape the array

In [230]: a = a.astype(int)   # type cast all elements from 'float64' to 'int'

In [231]: a
Out[231]:
array([[ 4,  1,  7],
       [ 4, 10,  1],
       [ 0,  7,  6],
       [ 1,  0,  3]])

In [232]: a.argsort()
Out[232]:
array([[1, 0, 2],
       [2, 0, 1],
       [0, 2, 1],
       [1, 0, 2]])

In [233]: a
Out[233]:
array([[ 4,  1,  7],
       [ 4, 10,  1],
       [ 0,  7,  6],
       [ 1,  0,  3]])
```

Type casting

- Type casting – *recap*

  Examples:

  - a = 3 + 4 -> creates an integer
  - b = 6 + 7. -> creates a floating point number (higher precision)

```
In [1]: a = plb.arange(5)

In [2]: a
Out[2]: array([0, 1, 2, 3, 4])

In [3]: a.dtype
Out[3]: dtype('int64')

In [4]: a = a*1.    # we force type casting by multiplying 'int' to a 'float'

In [5]: a
Out[5]: array([ 0.,  1.,  2.,  3.,  4.])

In [6]: a.dtype
Out[6]: dtype('float64')
```

Type casting

- Type casting
  - when forcing to assign a member in an array object with a different type, the assignment is completed, but no type conversion is performed on the array
  - as a result the newly assigned value may be truncated if assigning from a float to an int for ex.

```
Python

In [22]: g
Out[22]: array([ 1.,  2.,  2.,  2.,  2.,  2.,  2.,  2.,  2.,  2.])

In [23]: g.dtype
Out[23]: dtype('float64')

In [24]: g[4] = 10    # lets try to asing an 'int' to position 5

In [25]: g     # the type is not changed regardless of the mis-type assignment
Out[25]: array([  1.,   2.,   2.,   2.,  10.,   2.,   2.,   2.,   2.,   2.])

In [26]: g = g.astype(int)   # lets type cast 'g' to 'int'

In [27]: g.dtype
Out[27]: dtype('int64')

In [28]: g
Out[28]: array([ 1,  2,  2,  2, 10,  2,  2,  2,  2,  2])

In [29]: g[4] = 10.5  # lets try to asing a 'float64' to position 5

In [30]: g     # the type is not changed regardless of the mis-type assignment
Out[30]: array([ 1,  2,  2,  2, 10,  2,  2,  2,  2,  2])
```

Rounding

- Rounding
  - rounding with around():

    all numbers having digits after decimal point from x.0-x.4 are rounded to the lower number
    all numbers having digits after decimal point from x.5-x.9 are rounded to the higher number

```
Python

In [7]: b = plb.arange(1,2,0.1)

In [8]: b
Out[8]: array([ 1. ,  1.1,  1.2,  1.3,  1.4,  1.5,  1.6,  1.7,  1.8,  1.9])

In [9]: c = plb.around(b)   # round all in 'b' and assign it to 'c'

In [10]: c
Out[10]: array([ 1.,  1.,  1.,  1.,  1.,  2.,  2.,  2.,  2.,  2.])

In [11]: c.dtype
Out[11]: dtype('float64')

In [12]: c = c.astype(int)   # type casting to 'int'

In [13]: c.dtype
Out[13]: dtype('int64')
```

Rounding

- Rounding
  - rounding with floor() and ceil():

    sometimes there is a need for an alternative way of rounding such as:

    - floor: rounds to the lower number
    - ceil: rounds to the higher number

```
In [14]: d = plb.arange(1,2,0.1)

In [15]: d
Out[15]: array([ 1. ,  1.1,  1.2,  1.3,  1.4,  1.5,  1.6,  1.7,  1.8,  1.9])

In [16]: e = plb.around(d)   # normal rounding around mid-point '0.5' as expected

In [17]: e
Out[17]: array([ 1.,  1.,  1.,  1.,  1.,  2.,  2.,  2.,  2.,  2.])

In [18]: f = plb.floor(d)    # rounds all to the lower number

In [19]: f
Out[19]: array([ 1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.])

In [20]: g = plb.ceil(d)     # rounds all to the higher number

In [21]: g
Out[21]: array([ 1.,  2.,  2.,  2.,  2.,  2.,  2.,  2.,  2.,  2.])
```

Type info

- Type info – *for NumPy variables only*
  - to check the type of a scalar or an array we use:

    <scalar>.dtype or <array>.dtype

  - to check the type of a scalar or an array element we use:

    plb.dtype(<scalar>) or plb.dtype(<array[element]>)

  - to check how many bytes a type takes we use:

    plb.<type>().itemsize

  - to check how many bits a type takes we use:

    (plb.dtype(plb.<type>).itemsize)*8

  - to check the minimum, maximum values and the int type of a scalar variable or array element we use:

    plb.iinfo(plb.<type>)

Type info

- Type info – *for NumPy variables only* – Examples:
  - to check the type of a scalar or an array we use:

```
h = plb.int32(12)

h.dtype
dtype('int32')
```

or

```
i = plb.arange(4)

i.dtype
dtype('int64')
```

  - to check the type of a scalar or an array element we use:

```
plb.dtype(h)
dtype('int32')
```

or

```
plb.dtype(i[2])
dtype('int64')
```

  - to check how many bytes a type takes we use:

```
plb.int64().itemsize
8
```

  - to check how many bits a type takes we use:

```
(plb.dtype(plb.int64()).itemsize)*8
64
```

  - to check the minimum, maximum values and the int type of a scalar variable or array element we use:

```
plb.iinfo(plb.int64())
iinfo(min=-9223372036854775808, max=9223372036854775807, dtype=int64)
```

Type info

- Type info – *for NumPy variables only* – Examples:

  – to check the machine limits for floating point types namely resolution, minimum, maximum values, type we use:

```
plb.finfo(plb.float32())
finfo(resolution=1e-06, min=-3.4028235e+38, max=3.4028235e+38, dtype=float32)
```

  or we can use:

```
plb.finfo(plb.float32()).eps
1.1920929e-07
```

  where, the eps attribute shows the smallest representable positive number and is a float

  – we can also check for the speed of certain computations using different types:

```
In [31]: i = plb.arange(0,5,1)

In [32]: i
Out[32]: array([0, 1, 2, 3, 4])

In [33]: i.dtype
Out[33]: dtype('int64')

In [34]: %timeit i**6
100000 loops, best of 3: 2.81 µs per loop

In [35]: i = i.astype(plb.float64)

In [36]: i
Out[36]: array([ 0.,  1.,  2.,  3.,  4.])

In [37]: %timeit i**6
100000 loops, best of 3: 4.41 µs per loop
```

Type info

- Type info – *for NumPy variables only* – Examples:
  - we can also check for the speed of execution for larger pieces of code similar to the tic – toc functionality in Matlab:

Python

```
In [23]: # way to measure time laps in seconds:

In [24]: import time

In [25]: start_time = time.time()

In [26]: start_time
Out[26]: 1447704378.69137

In [27]: end_time = time.time()

In [28]: end_time
Out[28]: 1447704386.440553

In [29]: total_time = end_time - start_time

In [30]: total_time
Out[30]: 7.749182939529419

In [31]: print("start time is %s sec earlier than end time" % round(total_time,3))
start time is 7.749 sec earlier than end time
```

Masking data

- Masking data
  - it is sometimes necessary to only observe certain part of a large data set
  - in this way, one can exclude unwanted values like NaN or negative numbers
  - for that reason it is useful to mask the portion of the array, which is unwanted

```
In [38]: j = plb.array([12, 34, 41, 52]) # array

In [39]: k = plb.array([1, 0, 0, 1]) # creates a mask

In [40]: l = plb.ma.array(j, mask=k) # creates an array with masked values

In [41]: l[:]
Out[41]:
masked_array(data = [-- 34 41 --],
             mask = [ True False False  True],
       fill_value = 999999)

In [42]: l[0]
Out[42]: masked

In [43]: l[1]
Out[43]: 34

In [44]: l.mask[0]=False     # the mask for each array element can be changed

In [45]: l[0]
Out[45]: 12
```

Organizing arrays

- Organizing arrays
  - larger data has to be well organized
  - all fields need to have an appropriate description

```
In [46]: m = plb.zeros((3,), dtype=[('Store:','S4'),('count',int),('location',float)])

In [47]: m
Out[47]:
array([(b'', 0, 0.0), (b'', 0, 0.0), (b'', 0, 0.0)],
      dtype=[('Store:', 'S4'), ('count', '<i8'), ('location', '<f8')])

In [48]: m.dtype.names # to see only the name fields of each element
Out[48]: ('Store:', 'count', 'location')

In [49]: m['Store:'] = ['East','West','North'] # assign an array of store names

In [50]: m['count']=plb.arange(1,4,1) # assign a sequence of numbers

In [51]: # individual field indexing will access and asign each individual location:

In [52]: m[0]['location']=43.7896;  m[1]['location']=64.4321;  m[2]['location']=87.2315

In [53]: m[1]
Out[53]: (b'West', 2, 64.4321)

In [54]: m['location']
Out[54]: array([ 43.7896,  64.4321,  87.2315])

In [55]: m[1]['location'] = 5   # asigning an 'int' to a 'float' it will be recorded

In [56]: m[1]['count'] = 2.345  # assigning a 'float' to an 'int' it will be truncated

In [57]: m
Out[57]:
array([(b'East', 1, 43.7896), (b'West', 2, 5.0), (b'Nort', 3, 87.2315)],
      dtype=[('Store:', 'S4'), ('count', '<i8'), ('location', '<f8')])
```

Good coding practices

- Good coding practices

- use short and concise comments to describe your code when needed
- use explicit variable names so that it is easy to understand what they are
- make variable names and comments in English
- avoid changing global variables in local scope functions
- use clean style such as:
  - spaces in for loop and if statements
  - spaces after commas
  - spaces before = and after
- use the same conventions as everybody else before you
- do not hard code your variables to increase portability
- make your code readable and beautiful to look
- write a simple code
- Import / load only what you need in your workspace
- test your code!

After following these recommendations you will increase the reliability of your code!

You can read more at: https://www.python.org/dev/peps/pep-0008/#class-names