STSCI 4060

Lecture File 8

Xiaolong Yang
(xy44@cornell.edu)

The Basics of NumPy, SciPy

NumPy and SciPy

NumPy and SciPy are two powerful Python packages for efficient scientific computing.

NumPy is Python's fundamental scientific computing package:

- Specializes in numerical processing through multi-dimensional arrays (**ndarrays**).
- Allows element-by-element operations.
- Linear algebra formalism can be used without modifying the NumPy arrays before-hand.
- Modify array size dynamically.

SciPy is built on the NumPy array framework and can supply many advanced mathematical functions, such as

- integration,
- ordinary differential equation solvers,
- optimizations.

NumPy Arrays

NumPy arrays are similar to lists (which are highly flexible by storing different types of objects in one list), but normally only the same type of elements can be stored. Despite this limitation, NumPy arrays' operations are sped up significantly.

```
In [2]: import numpy as np
        ar=np.arange(1e6)
        theList=ar.tolist()

In [3]: %timeit ar*10

        100 loops, best of 3: 2.38 ms per loop

In [4]: %timeit theList*10

        10 loops, best of 3: 103 ms per loop
```

Useful NumPy Arrays Functions

There are many functions you can use to create an array in NumPy:

- numpy.array()
- numpy.arange()
- numpy.linspace()
- numpy.zeros()
- numpy.ones()
- numpy.eye()
- numpy.diag()
- numpy.random.rand()
- numpy.empty()
- numpy.tile()

Creating NumPy Arrays in IPython via array()

```
import numpy as np
ar1=np.array([0,5,10,15,20]) #one dimensional array converted from a list
ar2=np.array([[1,2,3], [4,5,6]]) #2D array
ar3=np.array([[[1,2,3], [4,5,6], [7,8,9]]])  #3D array
```

```
ar1
```

```
array([ 0,  5, 10, 15, 20])
```

```
ar2
```

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

```
ar3
```

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

```
ar2.shape  #to get the shape of the array
```

```
(2, 3)
```

```
ar3.shape
```

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

```
ar2.ndim   #to get the number of dimensions
```

```
2
```

Creating NumPy Arrays in IPython via arange()

```
# produces the integers from 0 to 9, not inclusive of 10
ar4=np.arange(10)  #the NumPy version of plain Python's range function
```

```
ar4
```

```
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

```
# start, end (exclusive), step size
ar5=np.arange(3,50,3)
```

```
ar5
```

```
array([ 3,  6,  9, 12, 15, 18, 21, 24, 27, 30, 33, 36, 39, 42, 45, 48])
```

```
ar6=np.arange(0,1,0.1)
```

```
ar6
```

```
array([ 0. ,  0.1,  0.2,  0.3,  0.4,  0.5,  0.6,  0.7,  0.8,  0.9])
```

```
ar6=np.arange(1,0,-0.1)  #negative step size
```

```
ar6
```

```
array([ 1. ,  0.9,  0.8,  0.7,  0.6,  0.5,  0.4,  0.3,  0.2,  0.1])
```

Creating NumPy Arrays in IPython via linspace()

```
# The linspace() function generates linear, evenly spaced elements
# between the start and the end.
ar7=np.linspace(10, 80, 50) # args - start, end, number of elements
```

```
ar7
```

```
array([ 10.        ,  11.42857143,  12.85714286,  14.28571429,
        15.71428571,  17.14285714,  18.57142857,  20.        ,
        21.42857143,  22.85714286,  24.28571429,  25.71428571,
        27.14285714,  28.57142857,  30.        ,  31.42857143,
        32.85714286,  34.28571429,  35.71428571,  37.14285714,
        38.57142857,  40.        ,  41.42857143,  42.85714286,
        44.28571429,  45.71428571,  47.14285714,  48.57142857,
        50.        ,  51.42857143,  52.85714286,  54.28571429,
        55.71428571,  57.14285714,  58.57142857,  60.        ,
        61.42857143,  62.85714286,  64.28571429,  65.71428571,
        67.14285714,  68.57142857,  70.        ,  71.42857143,
        72.85714286,  74.28571429,  75.71428571,  77.14285714,
        78.57142857,  80.        ])
```

```
ar8=np.linspace(0, 2.0/3, 10)
```

```
ar8
```

```
array([ 0.        ,  0.07407407,  0.14814815,  0.22222222,  0.2962963 ,
        0.37037037,  0.44444444,  0.51851852,  0.59259259,  0.66666667])
```

Creating NumPy Arrays via zeros() or ones()

```
# Produce a 5x8 array of zeros.
ar9=np.zeros((5,8))
```

```
ar9
```

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

```
# Produces a 2x3x4 array of 1's
ar10=np.ones((2,3,4))
```

```
ar10
```

```
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.]]])
```

```
ar10.ndim
```

```
3
```

Creating NumPy Arrays via eye() or diag()

```
# The eye() function produces an identity matrix
ar11=np.eye(4)
```

```
ar11
```

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

```
# The diag() function creates a diagonal array
ar12=np.diag((2,4,6,8,10)) # args: 2,4,6,8,10 are ordered diagnal elements
```

```
ar12
```

```
array([[ 2,  0,  0,  0,  0],
       [ 0,  4,  0,  0,  0],
       [ 0,  0,  6,  0,  0],
       [ 0,  0,  0,  8,  0],
       [ 0,  0,  0,  0, 10]])
```

Creating NumPy Arrays via random.rand() or random.randn()

```
# rand(m) produces m uniformly distributed random numbers with range 0 to 1
np.random.seed(100)   # Set seed
ar13=np.random.rand(10)
```

```
ar13
```

```
array([ 0.54340494,  0.27836939,  0.42451759,  0.84477613,  0.00471886,
        0.12156912,  0.67074908,  0.82585276,  0.13670659,  0.57509333])
```

```
# randn(m) produces m normally distributed (Gaussian) random numbers
ar14=np.random.randn(10)
```

```
ar14
```

```
array([-0.07961125, -0.88973148, -0.88179839,  0.01863895,  0.23784462,
        0.01354855, -1.6355294 , -1.04420988,  0.61303888,  0.73620521])
```

Creating NumPy Arrays via empty() or tile()

```
# The empty() creates an uninitialized array.
# It is a cheaper and faster way to allocate an array,
# rather than using ones() or zeros(), but you should only use it
# if you are sure that all the elements will be initialized later.
ar15=np.empty((2,3,4))
```

```
ar15
```

```
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.]]])
```

```
# The tile() function allows to construct an array from a smaller array
# by repeating it several times on the basis of a parameter.
np.tile(np.array([[1,2],[3,4]]), 3)
```

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

```
np.tile(np.array([[1,2],[3,4]]), (3,3))
```

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

NumPy Datatypes

- The default datatype in NumPy is **float** but it can also be integers, Boolean values, complex numbers, etc.
- You can use the **dtype** parameter to specify the type of contents of a numeric array, or to find out the datatype of your array.
- The datatype of a NumPy array (ndarray) may be changed by casting.

NumPy Datatypes -- Using the dtype Parameter

```
In [62]: ar16=np.array([-2,-1,0,1,2], dtype='float'); ar16
Out[62]: array([-2., -1.,  0.,  1.,  2.])

In [63]: ar16.dtype
Out[63]: dtype('float64')

In [64]: ar1.dtype
Out[64]: dtype('int64')

In [66]: ar17=np.array([1., 2,3,4])  # The default datatype in NumPy is float.

In [67]: ar17.dtype
Out[67]: dtype('float64')

In [68]: # In the case of strings, dtype is the length of the longest
         # string in the array.
         ar18=np.array(['Cornell', 'Statistics'])
         ar18.dtype
Out[68]: dtype('S10')
```

Casting NumPy Datatypes

The datatype of ndarray can be changed in much the same way as we cast in other languages such as Java or C/C++. This is done with the **astype()** function.

```
ar1.dtype
```

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

```
ar1
```

```
array([ 0,  5, 10, 15, 20])
```

```
ar1.astype(float)
```

```
array([  0.,   5.,  10.,  15.,  20.])
```

```
ar1.dtype # It does not change the dataype in place
```

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

```
ar17
```

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

```
ar17.astype(int)
```

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

```
ar17.dtype # It does not change the dataype in place
```

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

NumPy Array Indexing

NumPy arrays can be indexed in the same way as we index into any other Python sequences.

```
# print entire array, element 0, element 1, last element.
print ar4
ar4[0], ar4[1], ar4[-1]
```

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

(0, 1, 9)
```

```
# Multi-dimensional arrays are indexed using tuples of integers.
ar3=np.array([[1,2,3], [4,5,6], [7,8,9]]); ar3
```

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

```
ar3[2,2] # Index the third row and the third column
```

```
9
```

```
ar3[2] # Retrieve row 2 (or the third row)
```

```
array([7, 8, 9])
```

```
ar3[2,:]  # The same as ar3[2]; colon denotes all elements of the dimension
```

```
array([7, 8, 9])
```

```
ar3[:,0] # Retrieve coloum 0
```

```
array([1, 4, 7])
```

NumPy Array Indexing (cont'd)

3D array indexing

```
In [86]: ar19=np.array([[[1,2,3],
                         [4,5,6],
                         [7,8,9]],

                        [[10,11,12],
                         [13,14,15],
                         [16,17,18]]])

In [87]: ar19.ndim

Out[87]: 3

In [88]: ar19[1,1,1]

Out[88]: 14
```

```
In [89]: ar19[0]

Out[89]: array([[1, 2, 3],
                [4, 5, 6],
                [7, 8, 9]])

In [90]: ar19[:,1,:]

Out[90]: array([[ 4,  5,  6],
                [13, 14, 15]])

In [92]: ar19[1,1,:]

Out[92]: array([13, 14, 15])
```

Reverse a NumPy Array Using the ::-1 Idiom

```
In [95]: ar20=np.arange(15)

In [96]: ar20
Out[96]: array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14])

In [97]: ar20[::-1]
Out[97]: array([14, 13, 12, 11, 10,  9,  8,  7,  6,  5,  4,  3,  2,  1,  0])

In [101]: ar20
Out[101]: array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14])

In [102]: ar20=ar20[::-1]; ar20
Out[102]: array([14, 13, 12, 11, 10,  9,  8,  7,  6,  5,  4,  3,  2,  1,  0])
```

The ::-1 Idiom can Also be Used in the Multiple Dimensional Arrays

```
In [14]: mda=np.array([[[1,2,3],
                        [4,5,6],
                        [7,8,9]],

                       [[10,11,12],
                        [13,14,15],
                        [16,17,18]]])

In [15]: mda[::-1]

Out[15]: array([[[10, 11, 12],
                 [13, 14, 15],
                 [16, 17, 18]],

                [[ 1,  2,  3],
                 [ 4,  5,  6],
                 [ 7,  8,  9]]])

In [16]: mda[::-1, ::-1]

Out[16]: array([[[16, 17, 18],
                 [13, 14, 15],
                 [10, 11, 12]],

                [[ 7,  8,  9],
                 [ 4,  5,  6],
                 [ 1,  2,  3]]])

In [17]: mda[::-1, ::-1, ::-1]

Out[17]: array([[[18, 17, 16],
                 [15, 14, 13],
                 [12, 11, 10]],

                [[ 9,  8,  7],
                 [ 6,  5,  4],
                 [ 3,  2,  1]]])
```

NumPy Array Slicing

Arrays can be sliced using the following syntax: array[startIndex: endIndex: stepValue].

```
ar20=ar20[::-1]; ar20
```

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

```
ar20[1:10:3]
```

```
array([13, 10,  7])
```

```
# Obtain the first n-elements with arrar[:n]
# The implicit assumption is that startIndex=0, step=1
ar20[:5]
```

```
array([14, 13, 12, 11, 10])
```

```
# Obtain the element n until the end with array[n:]
ar20[10:]
```

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

```
# You can also slice an array with a stepValue alone array[::v]
ar20[::5]
```

```
array([14,  9,  4])
```

NumPy Array Indexing and Slicing

```
In [106]: ar21=np.array([[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],
                         [31,32,33,34,35,36]])

In [107]: ar21

Out[107]: array([[ 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, 35, 36]])

In [112]: ar21[1,4:6]

Out[112]: array([11, 12])

In [113]: ar21[3:,3:]

Out[113]: array([[22, 23, 24],
                 [28, 29, 30],
                 [34, 35, 36]])

In [114]: ar21[3::2,::2]

Out[114]: array([[19, 21, 23],
                 [31, 33, 35]])
```

Combining NumPy Array Slicing and Assignment

```
In [115]: # Combining NumPy Array Assignment and Slicing
          ar22=np.arange(6); ar22

Out[115]: array([0, 1, 2, 3, 4, 5])

In [116]: ar22[:3]=1

In [117]: ar22

Out[117]: array([1, 1, 1, 3, 4, 5])

In [120]: ar22[3:]=np.ones(3)

In [122]: ar22

Out[122]: array([1, 1, 1, 1, 1, 1])
```

Array Masking: Select or Filter out Array Elements

```
In [126]: ar23=np.array(['Harvard','Yale','Penn','','Columbia','','Princeton'])

In [127]: ar23

Out[127]: array(['Harvard', 'Yale', 'Penn', '', 'Columbia', '', 'Princeton'],
                dtype='|S9')

In [128]: ar23[ar23=='']='Cornell' # Replace missing values with a default value

In [129]: ar23

Out[129]: array(['Harvard', 'Yale', 'Penn', 'Cornell', 'Columbia', 'Cornell',
                 'Princeton'],
                dtype='|S9')

In [131]: ar24=np.random.random_integers(0,21,10); ar24

Out[131]: array([18, 13, 12,  3, 20, 10,  8,  8, 12, 21])

In [132]: evenNumbers=ar24[ar24%2==0]; evenNumbers  # Filter out even numbers

Out[132]: array([18, 12, 20, 10,  8,  8, 12])

In [133]: oddNumbers=ar24[ar24%2!=0]; oddNumbers    # Filter out odd numbers

Out[133]: array([13,  3, 21])
```

Numpy Array Basic Operations

Basic arithmetic operations (+, -, *, /, and **) work element-wise with scalar operands.

```
In [135]: ar25=np.arange(10)*10; ar25
Out[135]: array([ 0, 10, 20, 30, 40, 50, 60, 70, 80, 90])

In [136]: ar25/10
Out[136]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

In [138]: ar26=100-ar25; ar26
Out[138]: array([100,  90,  80,  70,  60,  50,  40,  30,  20,  10])

In [141]: ar27=ar25-100; ar27
Out[141]: array([-100,  -90,  -80,  -70,  -60,  -50,  -40,  -30,  -20,  -10])

In [142]: ar28=ar26-ar27; ar28      # The two arrays must have the same shape.
Out[142]: array([200, 180, 160, 140, 120, 100,  80,  60,  40,  20])
```

Numpy Array Basic Operations (cont'd)

Comparisons and logical operations are also element-wise. *The two arrays involved must have the same shape*.

```
In [165]: ar32=np.arange(1,9); ar32
Out[165]: array([1, 2, 3, 4, 5, 6, 7, 8])

In [166]: ar33=np.arange(9,1,-1); ar33
Out[166]: array([9, 8, 7, 6, 5, 4, 3, 2])

In [169]: ar32<ar33
Out[169]: array([ True,  True,  True,  True, False, False, False, False], dtype=boo
          l)

In [170]: ar34=np.array([True,True,False,True,False])
          ar35=np.array([True,False,True,True,True])
          np.logical_and(ar34, ar35)
Out[170]: array([ True, False, False,  True, False], dtype=bool)
```

Numpy Array Basic Operations (cont'd)

Other NumPy operations such as log, sin, cos, and exp are also element-wise.

```
In [178]: ar36=np.pi*np.linspace(0,1,11); ar36

Out[178]: array([ 0.        ,  0.31415927,  0.62831853,  0.9424778 ,  1.25663706,
                  1.57079633,  1.88495559,  2.19911486,  2.51327412,  2.82743339,
                  3.14159265])

In [179]: np.sin(ar36)

Out[179]: array([  0.00000000e+00,   3.09016994e-01,   5.87785252e-01,
                   8.09016994e-01,   9.51056516e-01,   1.00000000e+00,
                   9.51056516e-01,   8.09016994e-01,   5.87785252e-01,
                   3.09016994e-01,   1.22464680e-16])

In [182]: np.log(ar36[1:])

Out[182]: array([-1.15785521, -0.46470803, -0.05924292,  0.22843915,  0.45158271,
                  0.63390426,  0.78805494,  0.92158633,  1.03936937,  1.14472989])

In [183]: np.exp(ar36)

Out[183]: array([  1.        ,   1.36910777,   1.87445609,   2.5663324 ,
                   3.51358562,   4.81047738,   6.58606196,   9.01702861,
                  12.34528394,  16.90202417,  23.14069263])
```

Array Multiplication ≠ Matrix Multiplication

NumPy array multiplication is element-wise, i.e., the corresponding elements are multiplied together. For matrix multiplication, the dot operator is used.

```
In [143]: ar25*ar25
Out[143]: array([   0,  100,  400,  900, 1600, 2500, 3600, 4900, 6400, 8100])

In [144]: ar25.dot(ar25) #for 1-D arrays it is equivalent to inner product of vectors
Out[144]: 28500
```

```
In [149]: ar30=np.array([[1,2],
                         [3,4]])
          ar31=np.array([[5,6],
                         [7,8]])

In [150]: ar30*ar31
Out[150]: array([[ 5, 12],
                 [21, 32]])

In [151]: # For 2-D arrays it is equivalent
          # to matrix multiplication
          ar30.dot(ar31)
Out[151]: array([[19, 22],
                 [43, 50]])
```

Array Multiplication ≠ Matrix Multiplication

(Cont'd)

```
In [154]: np.dot(ar25,ar25) # Another way to use the dot operator

Out[154]: 28500

In [157]: # Convert an arry to a matrix first
          mat1=np.matrix(ar30)
          mat2=np.matrix(ar31)

In [158]: mat1*mat2

Out[158]: matrix([[19, 22],
                  [43, 50]])

In [161]: mat1.dot(mat2)  # You can also use the dot operator

Out[161]: matrix([[19, 22],
                  [43, 50]])
```

Transpose NumPy Arrays

```
In [184]: ar37=np.array([[1,2,3,4,5],[5,6,7,8,9]]); ar37

Out[184]: array([[1, 2, 3, 4, 5],
                 [5, 6, 7, 8, 9]])

In [185]: ar37.T

Out[185]: array([[1, 5],
                 [2, 6],
                 [3, 7],
                 [4, 8],
                 [5, 9]])

In [186]: np.transpose(ar37)

Out[186]: array([[1, 5],
                 [2, 6],
                 [3, 7],
                 [4, 8],
                 [5, 9]])
```

Common NumPy Array Statistical Operators

Operator	Function
amin(), amax()	Return the minimum or maximum of an array
percentile(a,q)	Compute the qth percentile of the data, q in [0, 100]
ptp()	Compute the range of values (maximum - minimum)
mean()	Compute the arithmetic mean
average()	Compute the weighted average
median()	Compute the median
std()	Compute the standard deviation
var()	Compute the variance
corrcoef()	Return Pearson product-moment correlation coefficients
correlate()	Cross-correlation of two 1-dimensional sequences
cov()	Estimate a covariance matrix, given data and weights
histogram()	Compute the histogram of a set of data.

Some NumPy Array Statistical Operator Examples

```
In [198]: ar38=np.array([1,2,3,4,5,6])

In [200]: np.amin(ar38)
Out[200]: 1

In [204]: ar38.mean()
Out[204]: 3.5

In [220]: np.random.seed(80)
          ar39=np.random.randint(0,10, size=(3,3));ar39
Out[220]: array([[6, 3, 2],
                 [3, 6, 7],
                 [7, 3, 1]])

In [221]: np.mean(ar39)  # the arithmetic mean of all the elements
Out[221]: 4.222222222222223

In [223]: np.std(ar39)
Out[223]: 2.1487866228681907

In [224]: np.mean(ar39, axis=0) # mean by column
Out[224]: array([ 5.33333333,  4.        ,  3.33333333])

In [225]: np.mean(ar39, axis=1)  # mean by row
Out[225]: array([ 3.66666667,  5.33333333,  3.66666667])
```

Some NumPy Array Statistical Operator Examples

```
In [227]: np.amax(ar39,axis=0)
Out[227]: array([7, 6, 7])

In [228]: np.amax(ar39,axis=1)
Out[228]: array([6, 7, 7])

In [238]: np.percentile(ar39,50)
Out[238]: 3.0

In [239]: np.median(ar39)
Out[239]: 3.0

In [236]: np.percentile(ar39,3, axis=0)
Out[236]: array([ 3.18,  3.  ,  1.06])

In [237]: np.histogram(ar39)
Out[237]: (array([1, 1, 0, 3, 0, 0, 0, 0, 2, 2]),
           array([ 1. ,  1.6,  2.2,  2.8,  3.4,  4. ,  4.6,  5.2,  5.8,  6.4,  7.
          ]))

In [240]: np.histogram(ar39,5)
Out[240]: (array([2, 3, 0, 0, 4]), array([ 1. ,  2.2,  3.4,  4.6,  5.8,  7. ]))
```

NumPy Array Logical Operators

The logical operators can be used for array comparisons.

- np.all()
- np.any()

```
In [314]: np.random.seed(100)
          ar40=np.random.randint(1,50, size=(5,5))
          ar40

Out[314]: array([[ 9, 25,  4, 40, 24],
                 [16, 49, 11, 31, 35],
                 [ 3, 35, 15, 35, 49],
                 [25, 16, 37, 44, 17],
                 [10, 30, 23,  3, 28]])

In [315]: np.any((ar40%5)==0)

Out[315]: True

In [316]: np.any((ar40%13)==0)

Out[316]: False

In [317]: np.any(ar40<60)

Out[317]: True
```

```
np.all((ar40%5)==0)

False

np.all((ar40<40))

False

np.all((ar40>=3))

True
```

NumPy Array Broadcasting

Broadcasting describes how NumPy treats arrays with different shapes during arithmetic operations under certain conditions. The smaller array is "broadcast" across the larger array so that they have compatible shapes. NumPy compares their shapes element-wise, starting with the trailing dimensions, and works its way forward. Two dimensions are compatible when they are **equal,** or one of them is **1**. The size of the resulting array is the maximum size along each dimension of the input arrays.

A	(2D array): 2 x 3	A	(3D array): 20 x 3 x 1
B	(1D array) : 3	B	(2D array) : 1 x 4
Result	(2D array) : 2 x 3	Result	(3D array) : 20 x 3 x 4
A	(2D array): 2 x 3	A	(3D array): 20 x 5 x 1
B	(1D array) : 1	B	(3D array) : 1 x 5 x 4
Result	(2D array) : 2 x 3	Result	(3D array) : 20 x 5 x 4

A	(2D array): 2 x 3	A	(3D array): 8 x 5 x 1
B	(1D array) : 2	B	(3D array) : 1 x 6 x 4
Result	?	Result	?

A NumPy Array Broadcasting Example

```
In [169]: ar41=np.ones((4,5)); ar41

Out[169]: array([[ 1.,  1.,  1.,  1.,  1.],
                 [ 1.,  1.,  1.,  1.,  1.],
                 [ 1.,  1.,  1.,  1.,  1.],
                 [ 1.,  1.,  1.,  1.,  1.]])

In [170]: ar42=np.array([10,20,30,40,50]); ar42

Out[170]: array([10, 20, 30, 40, 50])

In [171]: ar41+ar42

Out[171]: array([[ 11.,  21.,  31.,  41.,  51.],
                 [ 11.,  21.,  31.,  41.,  51.],
                 [ 11.,  21.,  31.,  41.,  51.],
                 [ 11.,  21.,  31.,  41.,  51.]])
```

ar41	(2D array): 4 x 5
ar42	(1D array) : 5
Result	(2D array) : 4 x 5

Flattening a Multi-Dimensional Array With ravel() or flatten()

```
In [335]: np.random.seed(110)
          ar43=np.random.randint(1,100, size=(4,4))
          ar43

Out[335]: array([[ 1, 62, 16, 90],
                 [48,  1, 81, 81],
                 [71, 99,  9,  6],
                 [33, 82, 98, 93]])

In [337]: ar43.ravel()

Out[337]: array([ 1, 62, 16, 90, 48,  1, 81, 81, 71, 99,  9,  6, 33, 82, 98,
          93])

In [338]: ar43.T.ravel()

Out[338]: array([ 1, 48, 71, 33, 62,  1, 99, 82, 16, 81,  9, 98, 90, 81,  6,
          93])

In [340]: ar43.flatten()

Out[340]: array([ 1, 62, 16, 90, 48,  1, 81, 81, 71, 99,  9,  6, 33, 82, 98,
          93])
```

ravel() vs. flatten()

The difference between ravel and flatten: flatten always returns a copy and ravel returns a view of the array whenever possible. This isn't visible in the printed output, but if you modify the array returned by ravel, it may modify (not always) the entries in the original array. If you modify the entries in an array returned from flatten this will never happen. Also, ravel is often faster since no memory is copied, but you have to be more careful about modifying the array it returns.

Reshape an Array

```
In [348]: ar44=np.arange(1,37); ar44

Out[348]: array([ 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,
                 35, 36])

In [349]: ar44.reshape(6,6)

Out[349]: array([[ 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, 35, 36]])

In [350]: ar44.reshape(4,9)

Out[350]: array([[ 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, 35, 36]])

In [352]: ar44.reshape(3,12)

Out[352]: array([[ 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, 35, 36]])
```

Resize an Array

There are two resize operators:

- numpy.ndarray.resize (new_sahpe), which resizes the ndarray in place;
- numpy.resize (a, new_shape), which returns a new array with new_shape.

```
In [353]: ar45=np.arange(10); ar45
Out[353]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

In [357]: ar46=np.resize(ar45, 15); ar46
Out[357]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 0, 1, 2, 3, 4])

In [358]: ar45
Out[358]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

In [369]: ar47=np.arange(10,20);

In [372]: ar47.resize(15)

In [371]: ar47
Out[371]: array([10, 11, 12, 13, 14, 15, 16, 17, 18, 19,  0,  0,  0,  0,  0])
```

Add a Dimension to an Array

```
In [376]: ar48=np.array([10,20,30,40,50]); ar48
Out[376]: array([10, 20, 30, 40, 50])

In [377]: ar48.shape
Out[377]: (5,)

In [378]: ar49=ar48[:,np.newaxis]; ar49.shape
Out[378]: (5, 1)

In [379]: ar49
Out[379]: array([[10],
                 [20],
                 [30],
                 [40],
                 [50]])
```

Sort an Array

A 2-dimensional array has two corresponding axes: the first axis (or x-axis) running vertically downwards across rows (axis 0), and the second axis (y-axis) running horizontally across columns (axis 1).

```
In [380]: ar50=np.array([[3,-1,10],[2,4,36],[5,-8,2]]); ar50

Out[380]: array([[ 3, -1, 10],
                 [ 2,  4, 36],
                 [ 5, -8,  2]])

In [383]: ar50.sort(axis=0); ar50 # sort along the x-axis in place

Out[383]: array([[ 2, -8,  2],
                 [ 3, -1, 10],
                 [ 5,  4, 36]])

In [384]: ar50.sort(axis=1); ar50 # sort along the y-axis in place

Out[384]: array([[-8,  2,  2],
                 [-1,  3, 10],
                 [ 4,  5, 36]])

In [387]: ar51=np.array([[3,-1,10],[2,4,36],[5,-8,2]])
          ar52=np.sort(ar51, axis=0); ar52  # sort along the x-axis out of place

Out[387]: array([[ 2, -8,  2],
                 [ 3, -1, 10],
                 [ 5,  4, 36]])

In [388]: ar51     # ar51 was not changed

Out[388]: array([[ 3, -1, 10],
                 [ 2,  4, 36],
                 [ 5, -8,  2]])
```

Some Applications

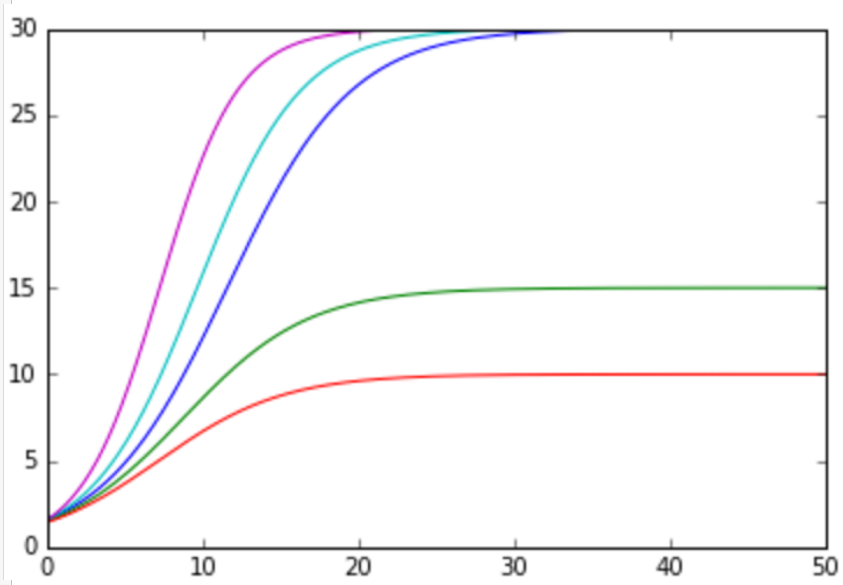
Describe a Growth Model

The **logistic growth** model is frequently used to represent growth of the population in an environment with limited resources and new products and/or technological innovations, which initially attract a small and quickly growing market but eventually reach a saturation point.

$$N(t) = \frac{a}{b + ce^{-rt}}$$

Describe a Growth Model (cont'd)

```
import numpy as np
import matplotlib.pyplot as plt
def make_logistic(r, a, b, c):
    def f_logistic(t):
        return a / (b + c * exp(-r * t))
    return f_logistic
r,a,c = 0.25,30.0,18.0
b1, b2, b3 = 1.0, 2.0, 3.0
r4,r5=0.3,0.4
logistic1 = make_logistic(r, a, b1, c)
logistic2 = make_logistic(r, a, b2, c)
logistic3 = make_logistic(r, a, b3, c)
logistic4 = make_logistic(r4, a, b1, c)
logistic5 = make_logistic(r5, a, b1, c)
tmax = 50
time_values = np.linspace(0, tmax, 500)
plt.plot(time_values, logistic1(time_values))
plt.plot(time_values, logistic2(time_values))
plt.plot(time_values, logistic3(time_values))
plt.plot(time_values, logistic4(time_values))
plt.plot(time_values, logistic5(time_values))
```



Describe a Growth Model (cont'd)

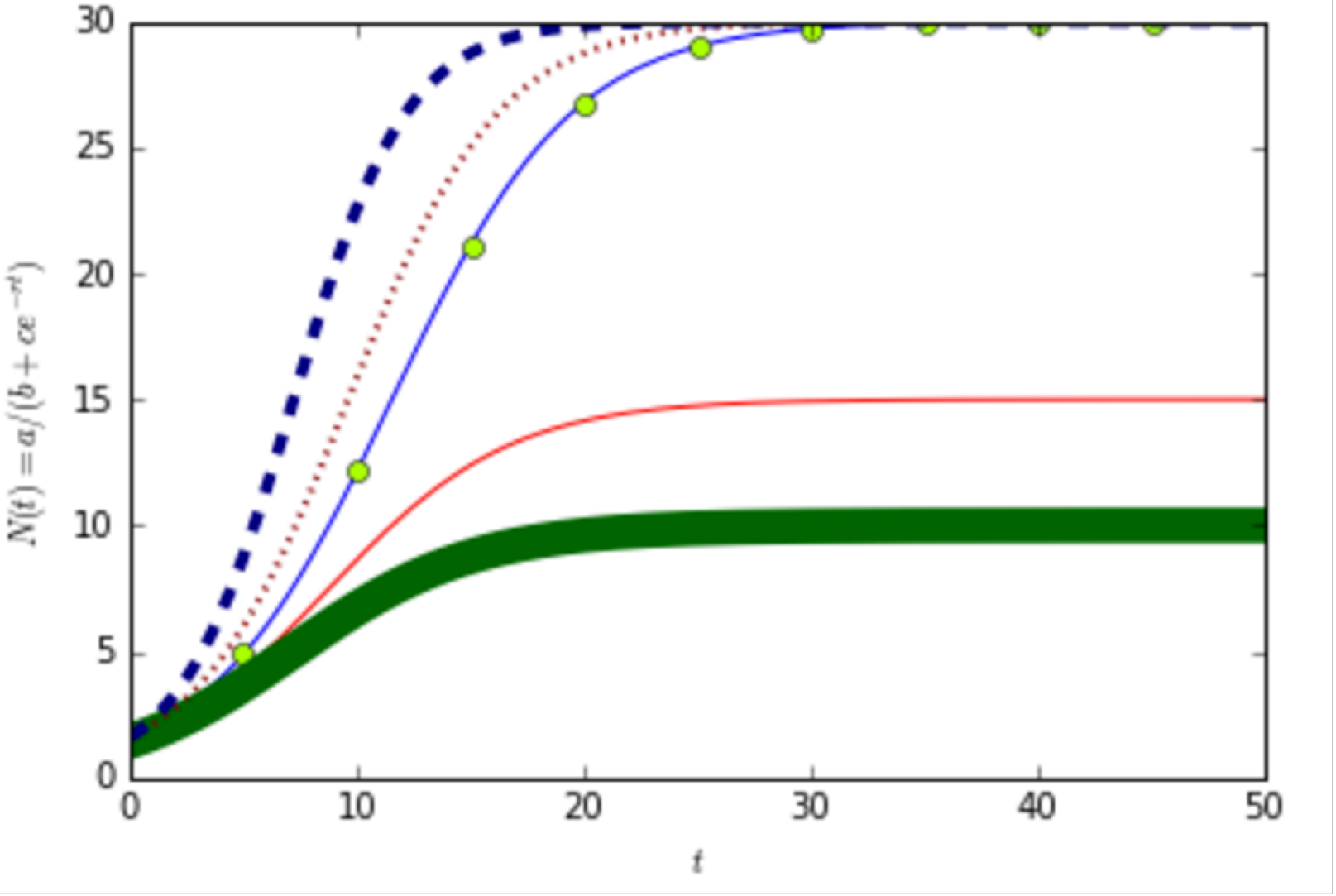
```
import numpy as np
import matplotlib.pyplot as plt
def make_logistic(r, a, b, c):
    def f_logistic(t):
        return a / (b + c * exp(-r * t))
    return f_logistic
r,a,c = 0.25,30.0,18.0
b1, b2, b3 = 1.0, 2.0, 3.0
r4,r5=0.3,0.4
logistic1 = make_logistic(r, a, b1, c)
logistic2 = make_logistic(r, a, b2, c)
logistic3 = make_logistic(r, a, b3, c)
logistic4 = make_logistic(r4, a, b1, c)
logistic5 = make_logistic(r5, a, b1, c)
tmax = 50
time_values = np.linspace(0, tmax, 500)
plt.plot(time_values, logistic1(time_values), marker='o',
         markevery=50,markerfacecolor='GreenYellow')
plt.plot(time_values, logistic2(time_values), color='Red')
plt.plot(time_values, logistic3(time_values),linewidth=10.5,
         color='DarkGreen', linestyle='-')
plt.plot(time_values, logistic4(time_values), linewidth=2.0,
         color='#8B0000', linestyle=':')
plt.plot(time_values, logistic5(time_values),linewidth=3.5,
         color=(0.0, 0.0, 0.5), linestyle='--')
xlabel('$t$')
ylabel('$N(t)=a/(b+ce^{-rt})$')
```

Symbol string	Line style
'-'	Solid (the default)
'--'	Dashed
':'	Dotted
'-.'	Dash-dot
'None',' ', or ''	Not displayed

marker	description
"."	point
","	pixel
"o"	circle
"v"	triangle_down
"^"	triangle_up
"<"	triangle_left
">"	triangle_right
"1"	tri_down
"2"	tri_up
"3"	tri_left
"4"	tri_right
"8"	octagon
"s"	square

Alias	Color
'b'	blue
'g'	green
'r'	red
'c'	cyan
'm'	magenta
'y'	yellow
'k'	black
'w'	white

Describe a Growth Model (cont'd)

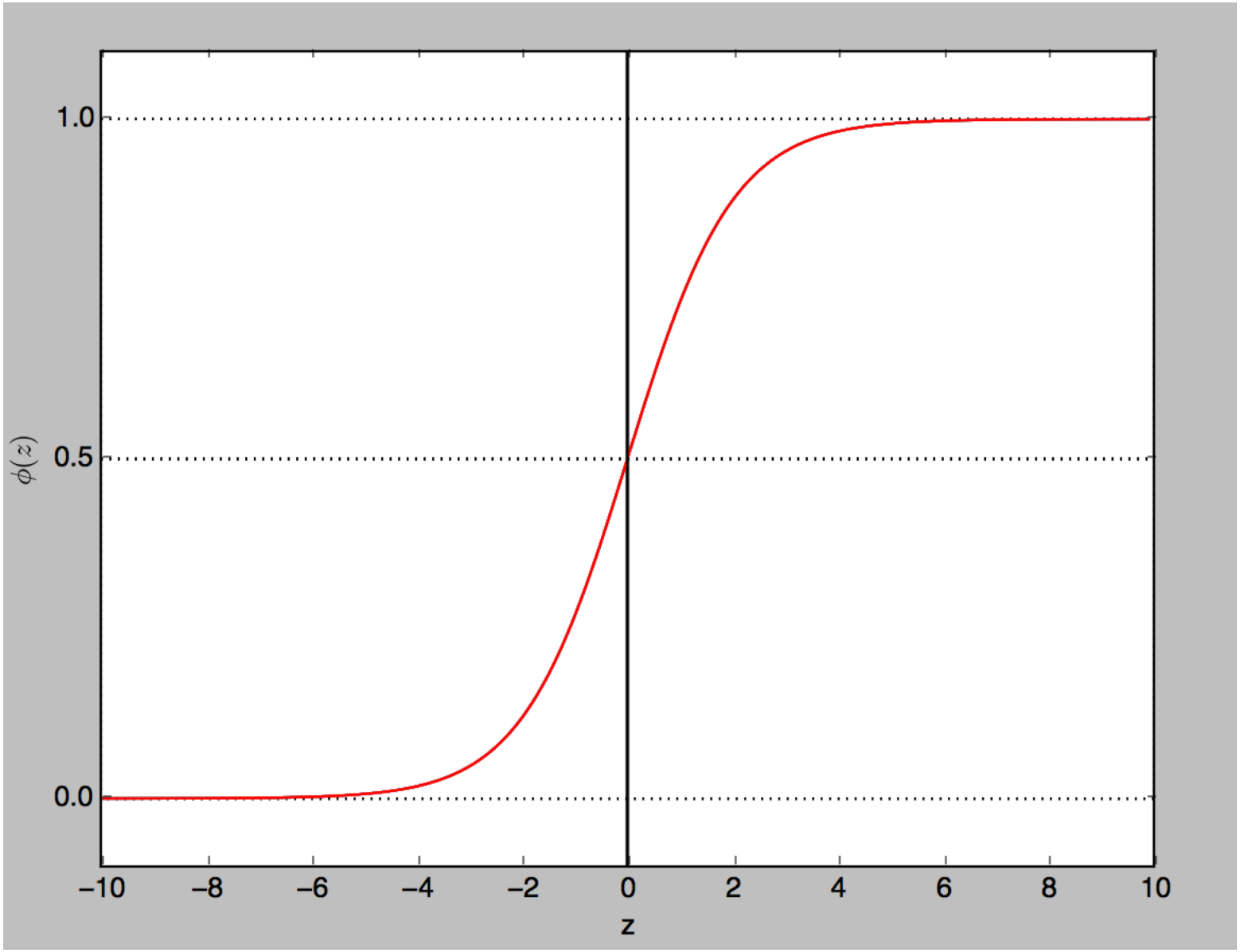


Plot a Logistic (sigmoid) Function

```
import matplotlib.pyplot as plt
import numpy as np
def sigmoid(z):
    return 1.0 / (1.0 + np.exp(-z))
z = np.arange(-10, 10, 0.1)
phi_z = sigmoid(z)
plt.plot(z, phi_z, color='r')
# add a vertical line across the axes,
# axvline(x=0, ymin=0, ymax=1,**kwargs)
plt.axvline(0.0, color='k')
# add a horizontal span (rectangle) across the axis,
# axhspan(ymin, ymax, xmin=0, xmax=1, **kwargs)
plt.axhspan(0.0, 1.0, facecolor='1.0', alpha=1.0, ls='dotted')
# add a horizontal line across the axis
plt.axhline(y=0.5, ls='dotted', color='k')
# set the y-limits of the current tick locations and labels
plt.yticks([0.0, 0.5, 1.0])
plt.xticks(np.linspace(-10,10,11))
plt.ylim(-0.1, 1.1)   # set the x limits of the current axes
plt.xlabel('z')       # Set the y axis label of the current axis
plt.ylabel('$\phi (z)$')
plt.show()
```

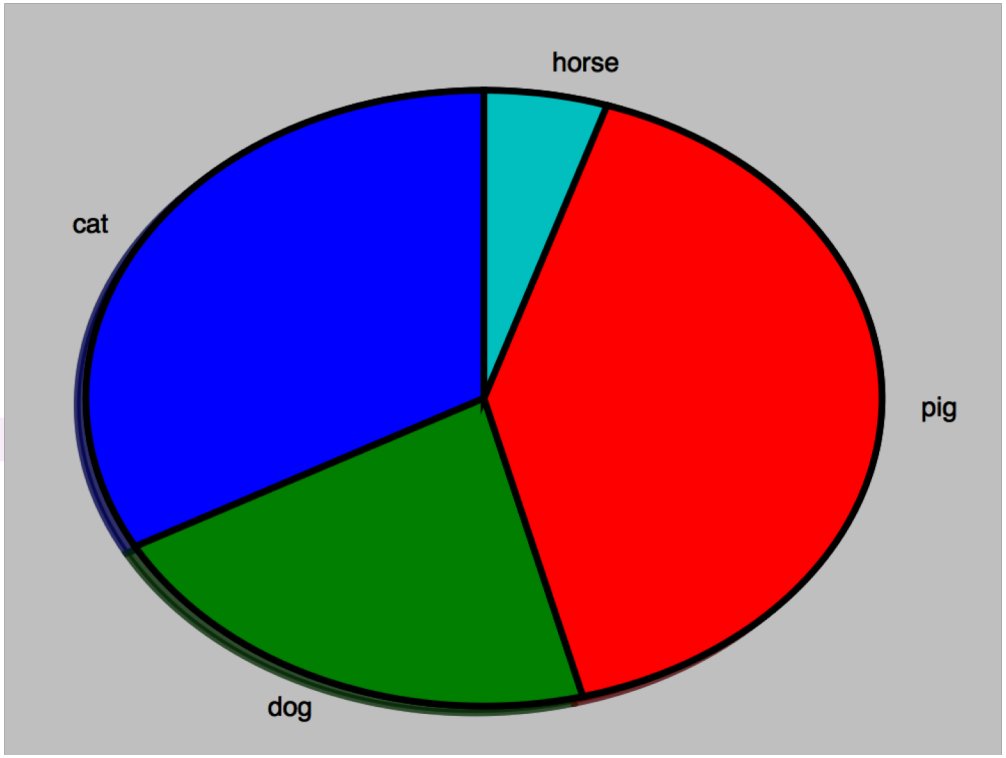
$$\phi(z) = \frac{1}{1+e^{-z}}$$

Plot a Logistic (sigmoid) Function (cont'd)



Plot a Pie Chart

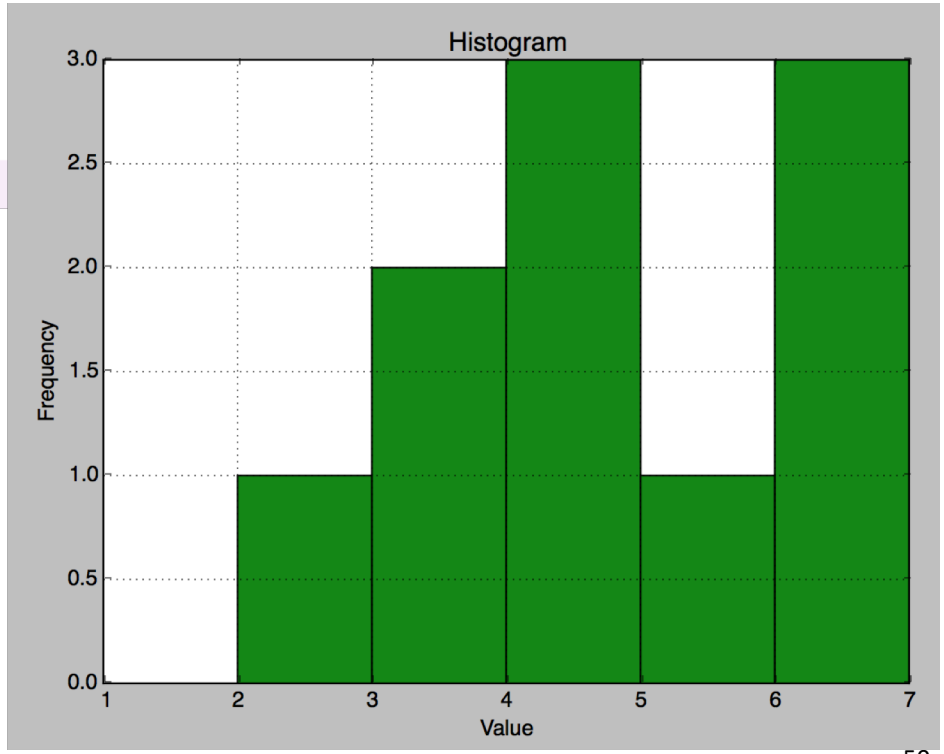
```
import matplotlib.pyplot as plt
import numpy as np

x=np.array([0.33,0.21,0.41,0.05])
plt.pie(x,
    explode=None,
    labels=np.array(['cat','dog','pig','horse']),
    colors=('b', 'g', 'r', 'c'),
    autopct=None,
    pctdistance=0.6,
    shadow=True,
    labeldistance=1.1,
    startangle=90,
    radius=1,
    counterclock=True,
    wedgeprops={'linewidth' : 3} ,
    textprops=None,
    center = (0, 0),
    frame =False )
plt.show()
```



Plot a Histogram

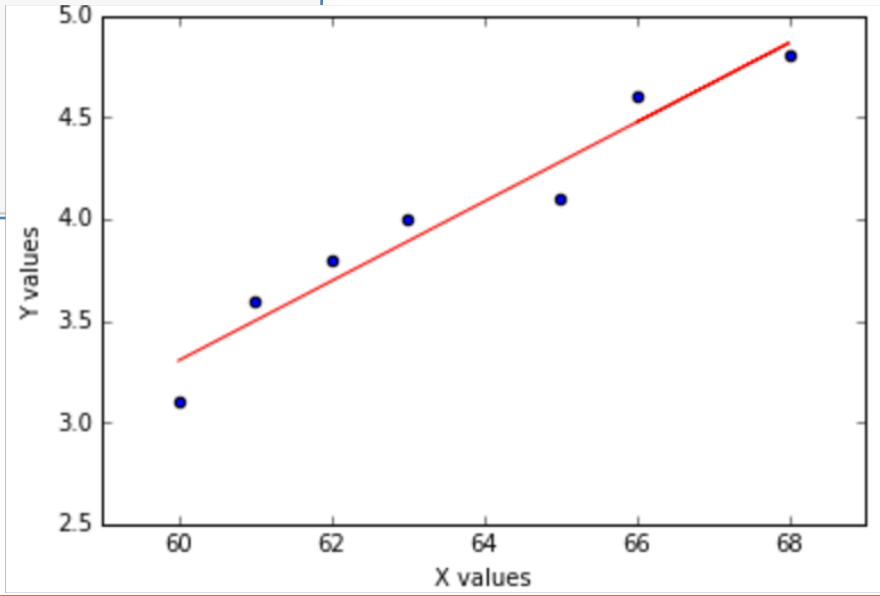
```
import numpy as np
import matplotlib.pyplot as plt

np.random.seed(80)
x=np.random.randint(1,8,10) #8, high, is exclusive
print 'Array x before sorted is ',x
x.sort()
print 'The sorted array x is ',x
plt.hist(x, bins=6, range=(1,7),facecolor='g', alpha=0.75)
plt.xlabel('Value')
plt.ylabel('Frequency')
plt.title('Histogram')
plt.grid(True)
plt.show()
```

Array x before sorted is [5 7 4 3 3 4 7 4 2 6]
The sorted array x is [2 3 3 4 4 4 5 6 7 7]



Redo the Previous Linear Regression Coding (1)

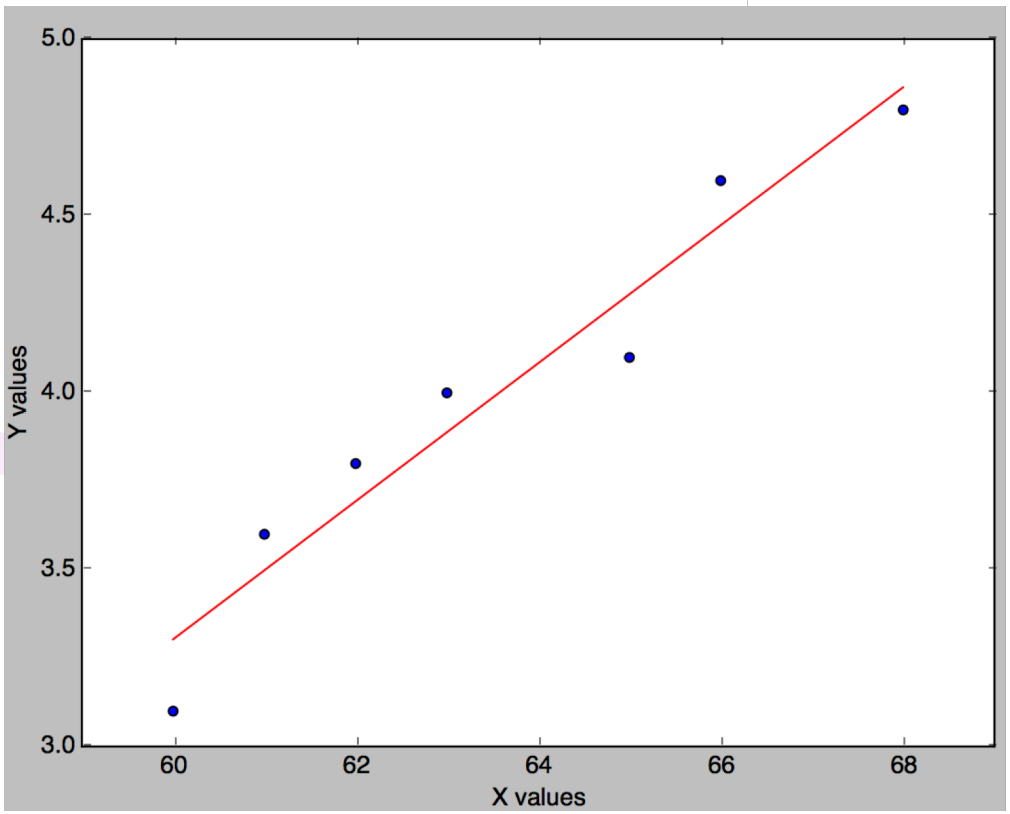
```
In [440]: import numpy as np
          import matplotlib.pyplot as plt
          from scipy.optimize import curve_fit
          x=np.array([60,61,62,63,65,68,66])
          y=np.array([3.1,3.6,3.8,4.0,4.1,4.8,4.6])
          def func(x, a, b):
              return a * x + b
          popt, pcov = curve_fit(func, x, y)
          popt

Out[440]: array([ 0.19511494, -8.40373563])

In [442]: Y=func(x, 0.19511494, -8.40373563)
          plt.plot(x,Y,'r-')
          plt.scatter(x,y)
          plt.xlabel('X values')
          plt.ylabel('Y values')
          plt.show()
```



Redo the Previous Linear Regression Coding (2)

```
 8 import numpy as np
 9 import matplotlib.pyplot as plt
10 from scipy import stats
11
12 x=np.array([60,61,62,63,65,68,66])
13 y=np.array([3.1,3.6,3.8,4.0,4.1,4.8,4.6])
14
15 slope, intercept, r_value, p_value, std_err=stats.linregress(x,y)
16 print 'Slope = ', slope
17 print 'Intercept = ', intercept
18 print 'R_value = ', r_value
19 print 'P_value = ', p_value
20 print 'STD_err = ', std_err
21
22 line=intercept+slope*x
23
24 plt.plot(x,line,'r-')
25 plt.scatter(x,y)
26 plt.xlabel('X values')
27 plt.ylabel('Y values')
28 plt.show()
```

```
Slope =  0.195114942529
Intercept =  -8.40373563218
R_value =  0.967955624169
P_value =  0.000347005979941
STD_err =  0.0226377698109
```



An Application of NumPy Arrays in Linear Algebra

NumPy arrays do not behave like matrices in linear algebra by default. You can use the built-in numpy.dot() and numpy.linalg.inv() functions. Or you can use the numpy.matrix() object instead.

$$3x + 6y - 5z = 12$$
$$x - 3y + 2z = -2$$
$$5x - y + 4z = 10$$

$$\underbrace{\begin{bmatrix} 3 & 6 & -5 \\ 1 & -3 & 2 \\ 5 & -1 & 4 \end{bmatrix}}_{A} \underbrace{\begin{bmatrix} x \\ y \\ z \end{bmatrix}}_{X} = \underbrace{\begin{bmatrix} 12 \\ -2 \\ 10 \end{bmatrix}}_{B}$$

$X = A^{-1}B$

```
In [425]: import numpy as np
          A=np.array([[3,6,-5],
                      [1,-3,2],
                      [5,-1,4]])
          B=np.array([12,-2,10])
          X=np.linalg.inv(A).dot(B)
          X

Out[425]: array([ 1.75,  1.75,  0.75])
```

So, we got x = 1.75, y = 1.75, z= 0.75.

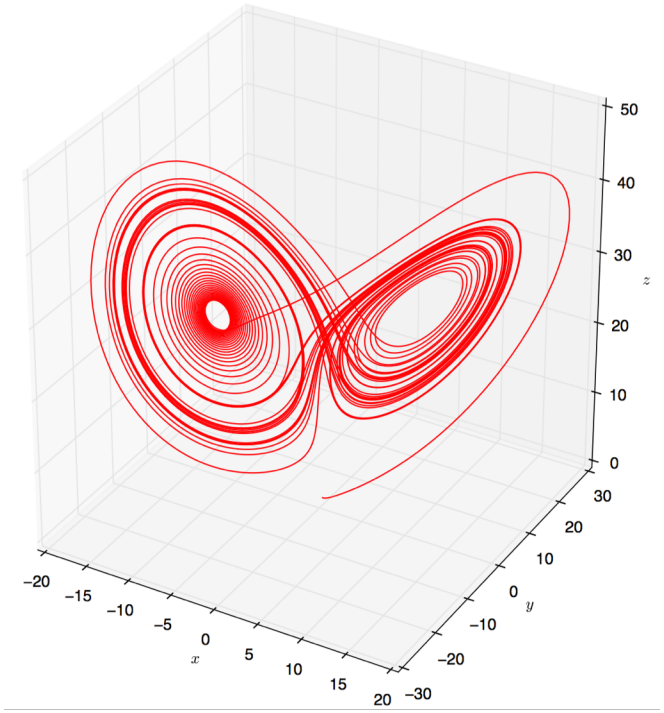
Or, use the numpy.matrix() function:

```
In [434]: import numpy as np
          a=np.matrix(A)

          b=np.matrix([[12],
                       [-2],
                       [10]])
          x=a**(-1)*b

          x

Out[434]: matrix([[ 1.75],
                  [ 1.75],
                  [ 0.75]])
```

An Example of Using NumPy, SciPy and Matplotlib in 3D Plotting

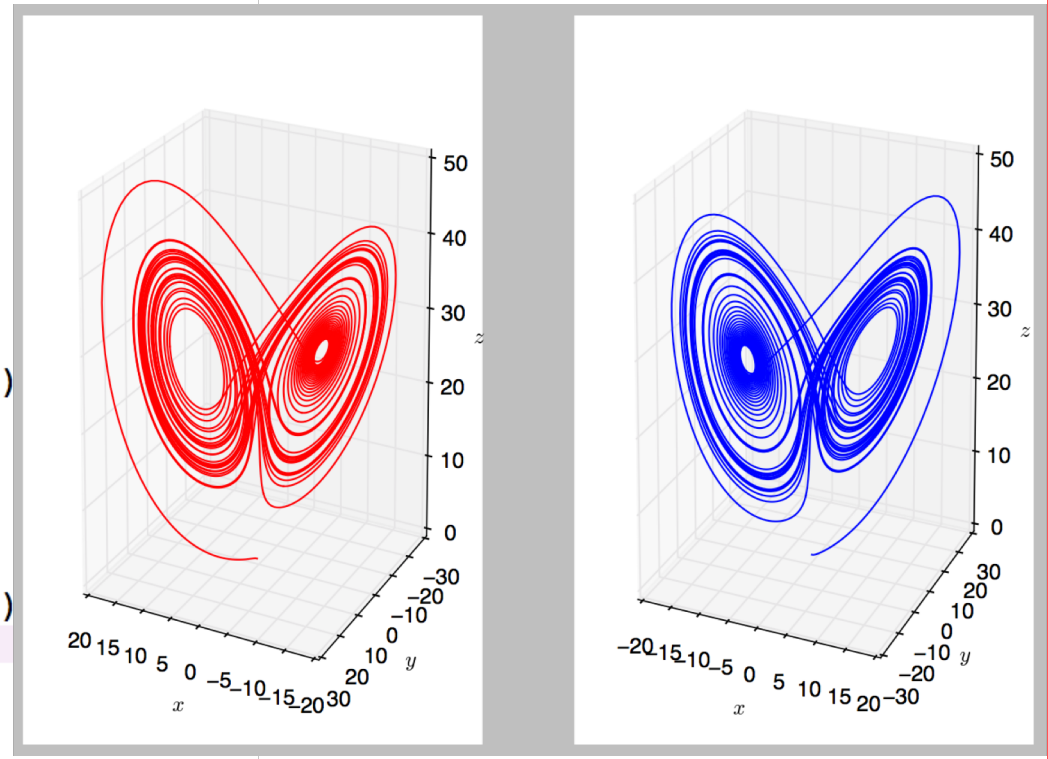
The **Lorenz system** is a system of ordinary differential equations first studied by Edward Lorenz. It is notable for having chaotic solutions for certain parameter values and initial conditions.

```
 8 import numpy as np
 9 import matplotlib.pyplot as plt
10 from scipy.integrate import odeint
11 from mpl_toolkits.mplot3d import Axes3D
12
13 def make_lorenz(sigma, r, b):
14     def func(statevec,t):
15         x,y,z=statevec
16         return [sigma*(y-x),r*x-y-x*z,x*y-b*z]
17     return func
18
19 lorenz_eq=make_lorenz(10.,28.,8./3.)#common: 10, 8/3
20
21 tmax=50
22 tdelta=0.005
23 tvalues=np.arange(0,tmax,tdelta)
24 ic=np.array([0.0,1.0,0.0])
25 sol=odeint(lorenz_eq, ic, tvalues)
26
27 x,y,z=np.array(zip(*sol))
28
29 fig=plt.figure(figsize=(10,10))
30 ax=fig.add_subplot(111, projection='3d')
31 ax.plot(x,y,z,lw=1,color='red')
32 ax.set_xlabel('$x$')
33 ax.set_ylabel('$y$')
34 ax.set_zlabel('$z$')
35 plt.show()
```



An Example of Using NumPy, SciPy and Matplotlib in 3D Plotting

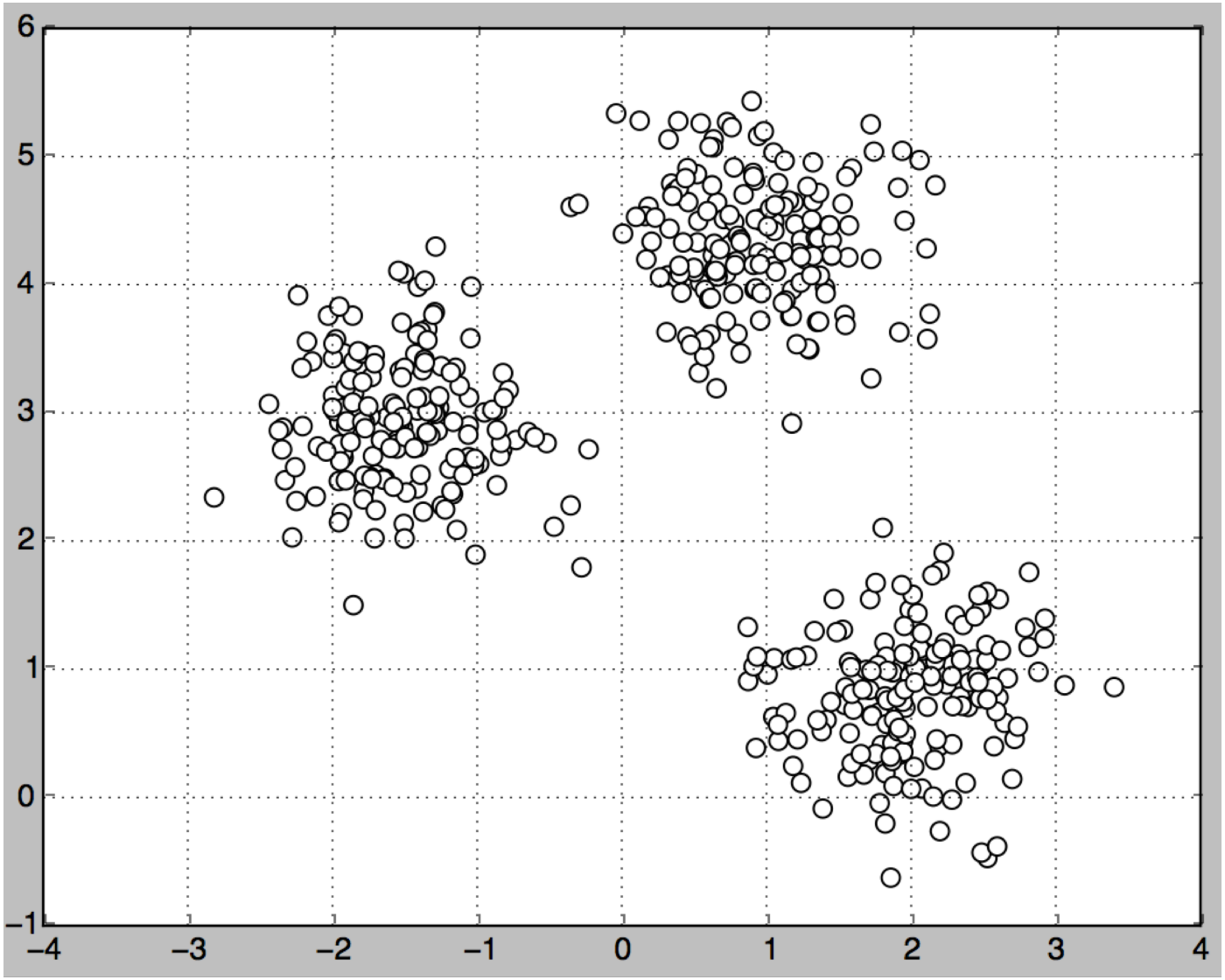
```
7  import numpy as np
8  import matplotlib.pyplot as plt
9  from scipy.integrate import odeint
10 from mpl_toolkits.mplot3d import Axes3D
11
12 def make_lorenz(sigma, r, b):
13     def func(statevec,t):
14         x,y,z=statevec
15         return [sigma*(y-x),r*x-y-x*z,x*y-b*z]
16     return func
17
18 lorenz_eq=make_lorenz(10.,28.,8./3.) #common: 10, 8/3
19
20 tmax=50
21 tdelta=0.005
22 tvalues=np.arange(0,tmax,tdelta)
23 ic=np.array([0.0,1.0,0.0])
24 sol=odeint(lorenz_eq, ic, tvalues)
25
26 x,y,z=np.array(zip(*sol))
27
28 fig=plt.figure(figsize=(10,7))
29 ax1=fig.add_subplot(121, projection='3d')
30 ax1.plot(x,y,z,lw=1,color='red')
31 ax1.set_xlabel('$x$')
32 ax1.set_ylabel('$y$')
33 ax1.set_zlabel('$z$')
34
35 ax2=fig.add_subplot(122, projection='3d')
36 ax2.plot(x,y,z,lw=1,color='blue')
37 ax2.set_xlabel('$x$')
38 ax2.set_ylabel('$y$')
39 ax2.set_zlabel('$z$')
40 plt.show()
```



Cluster Plotting: Code

```
import matplotlib.pyplot as plt
from sklearn.datasets import make_blobs

#generate an array of samples X of shape[n_sample, n_features]
#y,an array of integer labels for cluster membership of each sample
X, y = make_blobs(n_samples=500,
     centers=3,    #number of centers to generate
     cluster_std=0.5, #standard deviation of the clusters
     shuffle=True,
     random_state=0) #the seed used by the random number generator

plt.scatter(X[:,0],
            X[:,1],
            c='white',
            marker='o',
            s=50)       # s specifies the area of the marker

plt.grid()
plt.show()
```

Cluster Plotting: Output



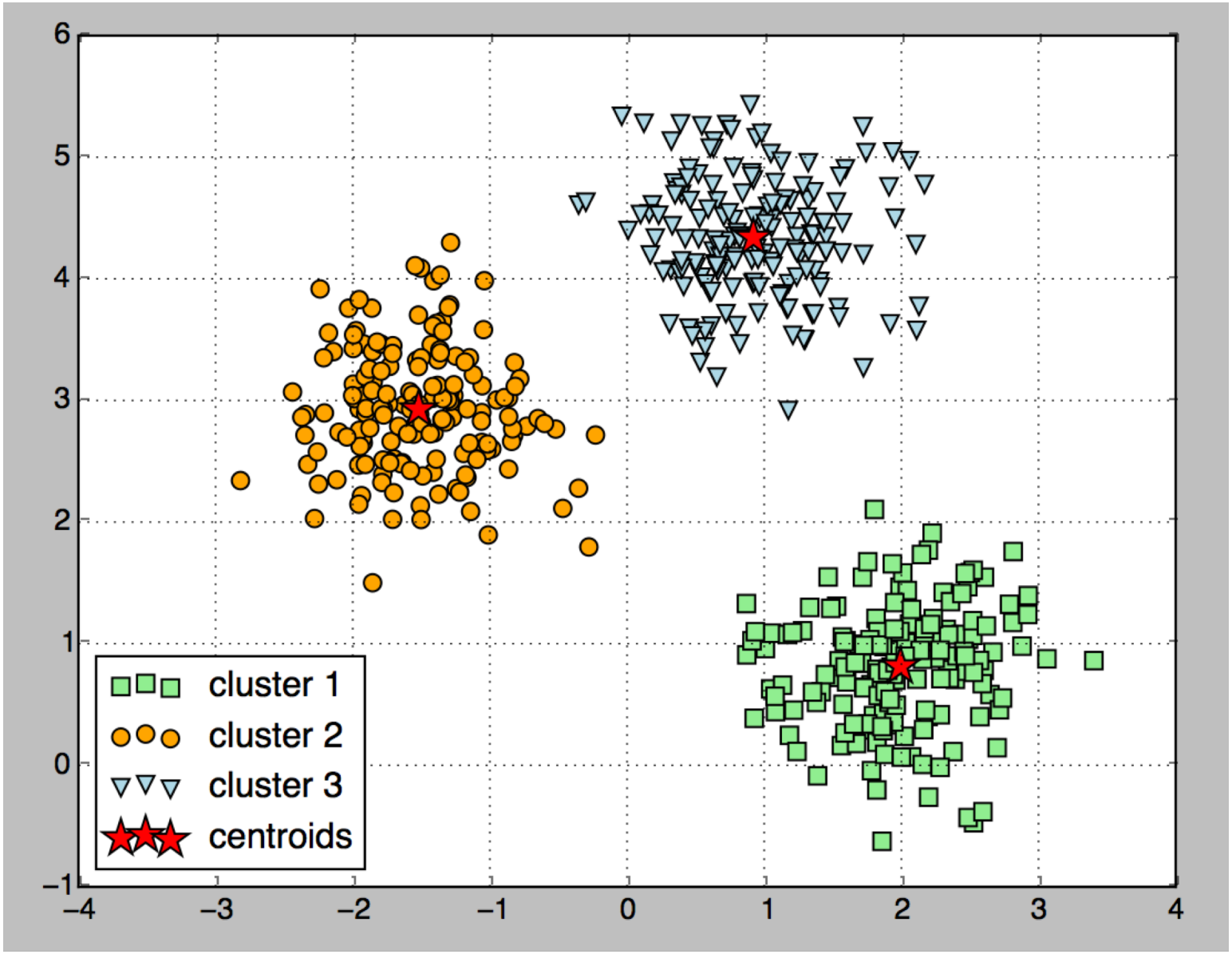
Cluster Analysis: K-Means

```
import matplotlib.pyplot as plt
from sklearn.datasets import make_blobs
from sklearn.cluster import KMeans

X,y=make_blobs(n_samples=500,
               n_features=2,
               centers=3,
               cluster_std=0.5,
               shuffle=True,
               random_state=0)
```

```
km = KMeans(n_clusters=3,
    init='random',
    n_init=10,
    max_iter=300,
    tol=1e-04,
    random_state=0)
y_km = km.fit_predict(X)

plt.scatter(X[y_km==0,0],
    X[y_km ==0,1],
    s=50,
    c='lightgreen',
    marker='s',
    label='cluster 1')
plt.scatter(X[y_km ==1,0],
    X[y_km ==1,1],
    s=50,
    c='orange',
    marker='o',
    label='cluster 2')
```

```
plt.scatter(X[y_km ==2,0],
    X[y_km ==2,1],
    s=50,
    c='lightblue',
     marker='v',
    label='cluster 3')
plt.scatter(km.cluster_centers_[:,0],
    km.cluster_centers_[:,1],
    s=250,
    marker='*',
    c='red',
    label='centroids')
plt.legend(loc=3)
plt.grid()
plt.show()
```

Cluster Analysis: K-Means Output

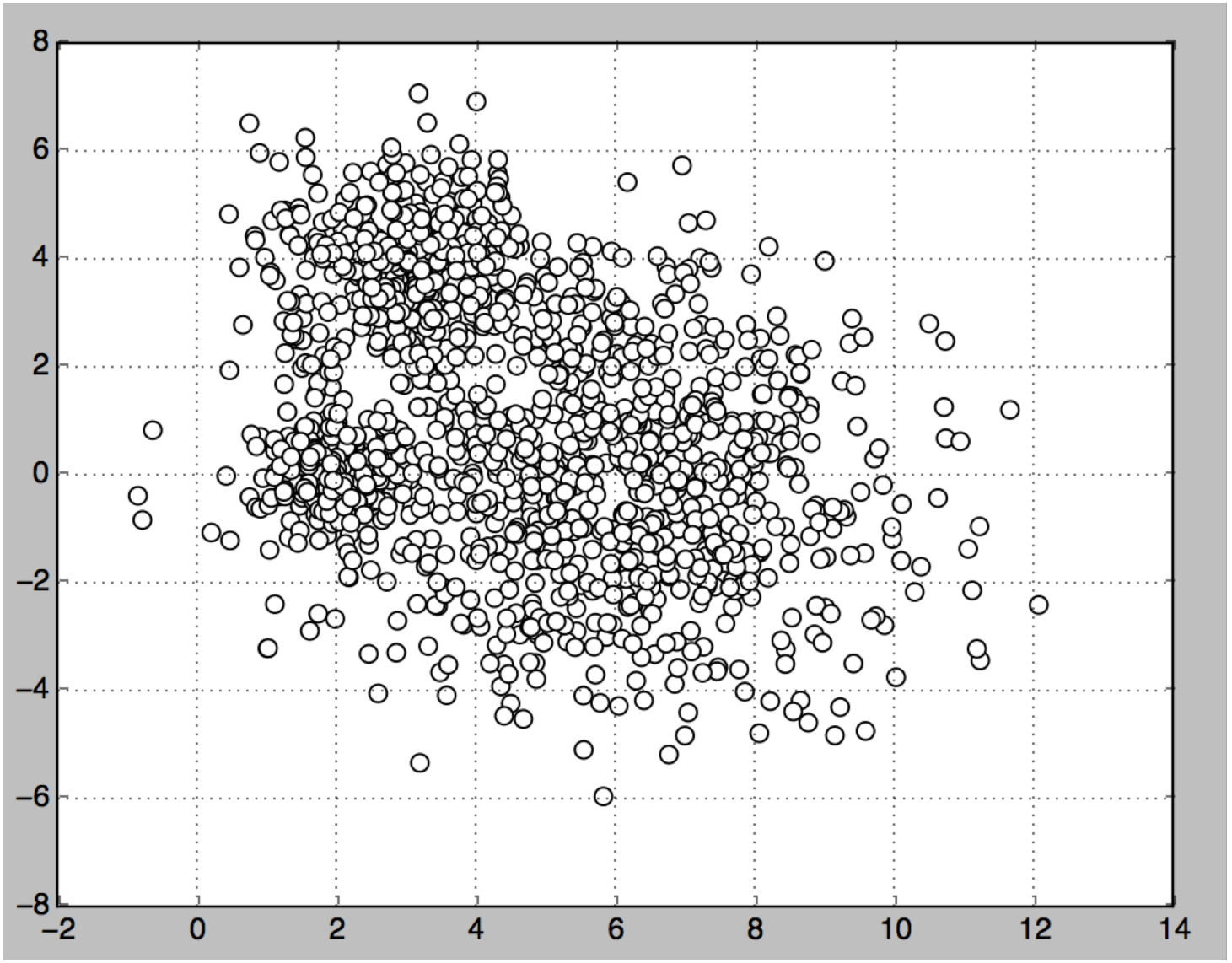


Cluster Plotting from a Data File

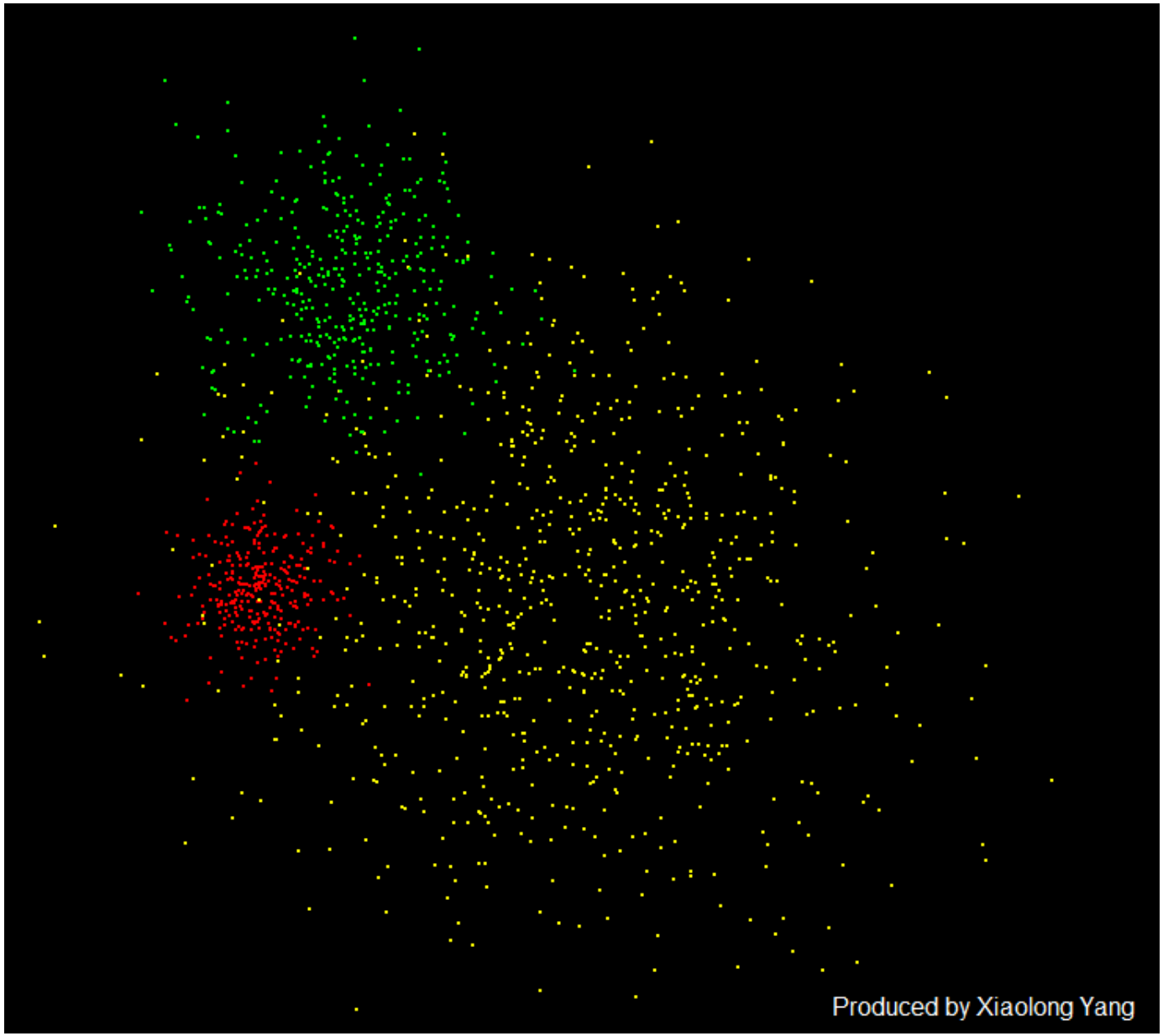
```
import matplotlib.pyplot as plt
import numpy as np
readin = open("/Users/xy44/Documents/STSCI4060/File5/unequal.txt")
lx=[]
ly=[]
for aline in readin:
    splits = aline.split()
    lx.append(splits[1])
    ly.append(splits[2])

x=np.array(lx).astype(float)
y=np.array(ly).astype(float)
plt.scatter(x,
            y,
            c='white',
            marker='o',
            s=50)       # s specifies

plt.grid()
plt.show()
```

```
1 2.9024114753 -0.03995751
1 2.1982884275 -0.541658827
1 3.1191471825 -0.312116147
1 2.2568288541 -0.043304558
1 1.7029106333 0.015945409
1 1.6311007139 -0.125069587
1 2.3425023824 -0.402079066
1 1.6278594599 -0.397751411
1 2.1703552747 -0.150254902
1 1.3250767422 0.2163524305
1 2.6528581213 0.7125635052
1 1.7920994903 0.8071902711
1 1.471136788 -0.474166336
1 2.47682382 0.1959894708
1 1.9619293604 0.6102784644
```

Cluster Plotting from a Data File: the Output



Real Clusters

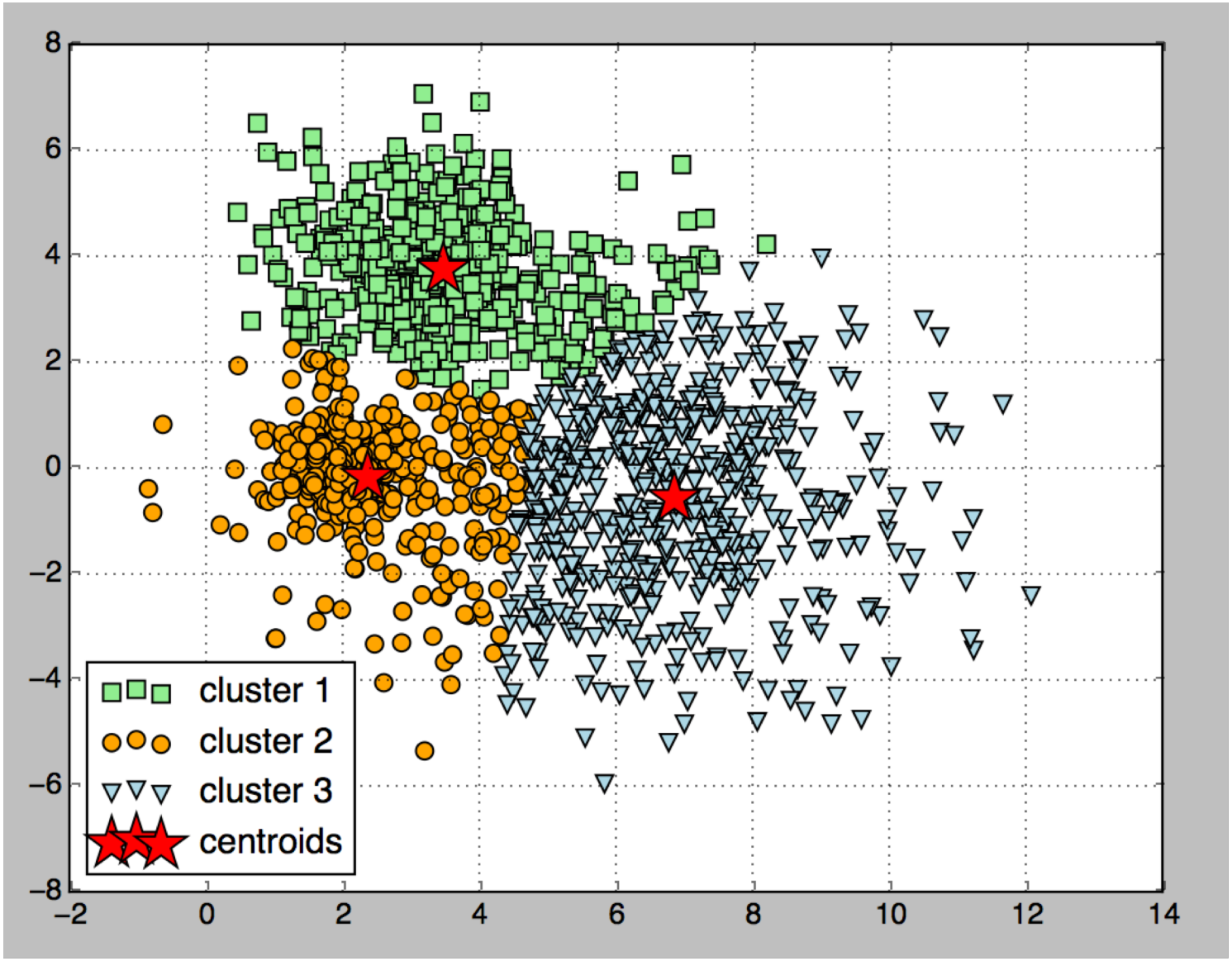


K-Means Cluster Analysis: Another Example-Data from a File

```
import matplotlib.pyplot as plt
import numpy as np
from sklearn.cluster import KMeans

readin = open("/Users/xy44/Documents/STSCI4060/File5/unequal.txt")
lx=[]
ly=[]
for aline in readin:
    splits = aline.split()
    lx.append(splits[1])
    ly.append(splits[2])
lengthList=len(lx)
x=np.array(lx).astype(float)
y=np.array(ly).astype(float)
```

```
X=np.empty((lengthList,2))
X[:,0]=x
X[:,1]=y
km = KMeans(n_clusters=3,
    init='random',
    n_init=10,
    max_iter=300,
    tol=1e-04,
    random_state=0)
y_km = km.fit_predict(X)
plt.scatter(X[y_km==0,0],
    X[y_km ==0,1],
    s=50,
    c='lightgreen',
    marker='s',
    label='cluster 1')
plt.scatter(X[y_km ==1,0],
    X[y_km ==1,1],
    s=50,
    c='orange',
    marker='o',
    label='cluster 2')
plt.scatter(X[y_km ==2,0],
    X[y_km ==2,1],
    s=50,
    c='lightblue',
     marker='v',
    label='cluster 3')
plt.scatter(km.cluster_centers_[:,0],
    km.cluster_centers_[:,1],
    s=500,
    marker='*',
    c='red',
    label='centroids')

plt.legend(loc=3)
plt.grid()
plt.show()
```

K-Means Cluster Analysis: Another Example-Data from a File Output



The Lorenz Equations

Edward Lorenz developed a simplified mathematical model for atmospheric convection. The model is a system of three ordinary differential equations now known as the Lorenz equations:

$$dx/dt = \mathbf{sigma}(y-x)$$
$$dy/dt = x(\mathbf{r}-z)-y$$
$$dz/dt = xy-\mathbf{b}z$$