Data Science: Machine Learning & and Deep Learning with Python

Data Analysis

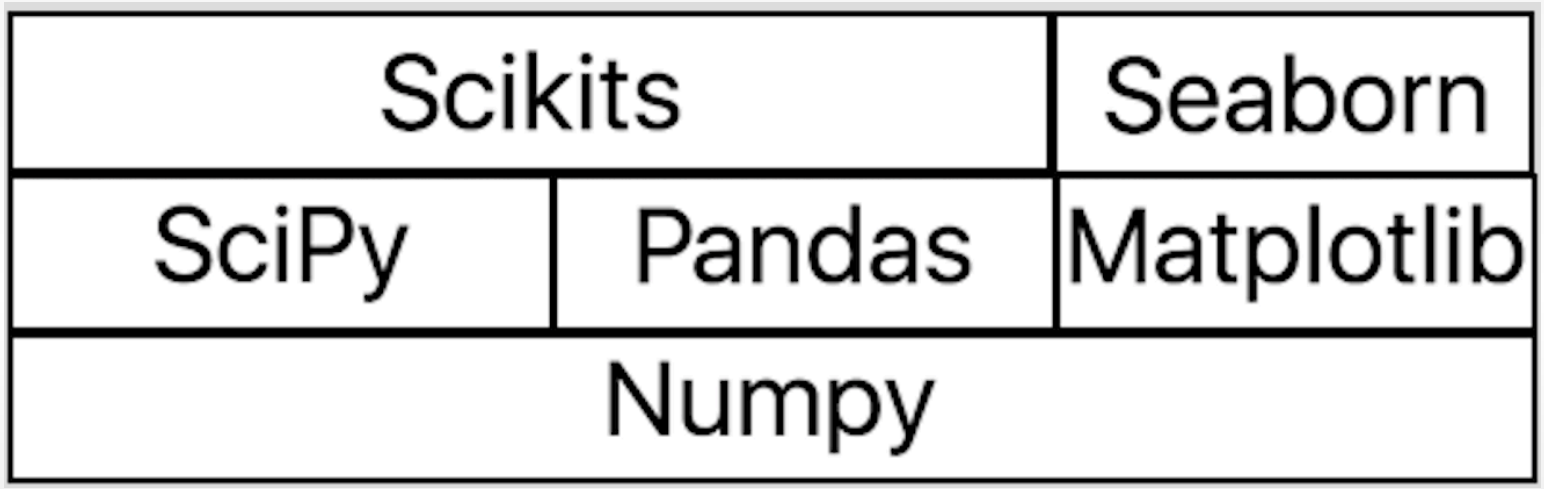
Data Analysis

Data analysis is a process of inspecting, cleansing, transforming, and modeling data with the goal of discovering useful information, informing conclusions, and supporting decision-making.

Data analysis has multiple facets and approaches, encompassing diverse techniques under a variety of names, while being used in different business, science, and social science domains.

In today's business, data analysis is playing a role in making decisions more scientific and helping the business achieve effective operation

Packages



Numpy

Python is a fabulous language

- Easy to extend
- Great syntax which encourages easy to write and maintain code
- Incredibly large standard-library and third-party tools

No built-in multi-dimensional array (but it supports the needed syntax for extracting elements from one)

NumPy provides a **fast** built-in object (ndarray) which is a multi-dimensional array of a homogeneous data-type.

https://www.scipy.org

SciPy

Scientific Packages

- Special Functions (scipy.special)
- Signal Processing (scipy.signal)
- Image Processing (scipy.ndimage)
- Fourier Transforms (scipy.fftpack)
- Optimization (scipy.optimize)
- Numerical Integration (scipy.integrate)
- Linear Algebra (scipy.linalg)
- ….

Available at www.scipy.org

Open Source BSD Style License

Pandas

Python Library to provide data analysis features similar to : R, MATLB, SAS

Rich data structures and functions to make working with data structure fast, easy and expressive.

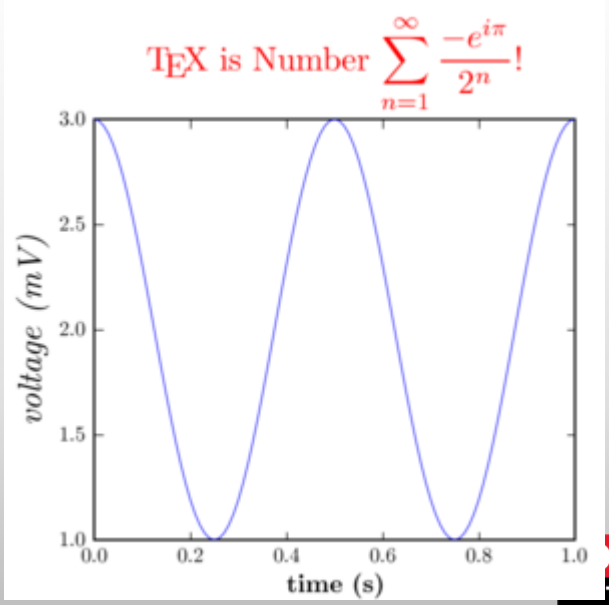
It is built on top of NumPy which provides it agility

Key components provided by Pandas : Two new data structures to Python

- Series
- DataFrame

Matplotlib

- Requires NumPy extension. Provides powerful plotting commands.
- http://matplotlib.sourceforge.net



Seaborn

Statistical plotting library

Built on Matplotlib

Great styles

Works great with NumPy arrays and Pandas Dataframes

Scikits

add-on toolkits that complement Scipy

The SciKits cover a broad spectrum of application domains, including:

- Financial computation
- Audio processing
- Geosciences
- Computer vision
- Engineering
- Machine learning
- Medical computing
- Bioinformatics

https://scikits.appspot.com/scikits

Overview

Python is a fabulous language

- Easy to extend
- Great syntax which encourages easy to write and maintain code
- Incredibly large standard-library and third-party tools

No built-in multi-dimensional array (but it supports the needed syntax for extracting elements from one)

NumPy provides a **fast** built-in object (ndarray) which is a multi-dimensional array of a homogeneous data-type.

N-D Array

- N-dimensional array of rectangular data
- Element of the array can be C-structure or simple data-type.
- Fast algorithms on machine data-types (int, float, etc.)

```
import numpy as np
from pylab import *

a = np.array([1, 4, 5, 8], float)
b = np.array([1, 2, 3, 4], float)
a=a*b
print(a[1])
plot(a,b)
```

Universal Functions

Functions that operate element-by-element and return result

Fast-loops registered for each fundamental data-type

- $\sin(x) = [\sin(x_i)\ i=0..N]$
- $x+y = [x_i + y_i\ i=0..N]$

```
import numpy as np
from pylab import *

a = np.array([1, 4, 5, 8], float)
b = np.array([1, 2, 3, 4], float)

c=np.add(a,b)

print(c)
```

Performance

```
In [20]: %%timeit
    ...: l=range(100000)
    ...: total = 0
    ...: for n in l:
    ...:     total +=n*n
    ...:
100 loops, best of 3: 7.38 ms per loop

In [21]: %%timeit
    ...: n=numpy.arange(100000)
    ...: numpy.sum(n*n)
    ...:
The slowest run took 24.88 times longer than the fastest. This could mean
that an intermediate result is being cached.
10000 loops, best of 3: 174 µs per loop
```

NumPy Array

A NumPy array is a homogeneous collection of “items” of the same “data-type” (dtype)

```
>>> import numpy as N
>>> a = N.array([[1,2,3],[4,5,6]],float)
>>> print a
[[1. 2. 3.]
 [4. 5. 6.]]

>>> print a.shape, "\n", a.itemsize
(2, 3)
8

>>> print a.dtype, a.dtype.type
'<f8' <type 'float64scalar'>

>>> type(a[0,0])
<type 'float64scalar'>

>>> type(a[0,0]) is type(a[1,2])
```

Introducing NumPy Arrays

SIMPLE ARRAY CREATION

```
>>> a = array([0,1,2,3])
>>> a
array([0, 1, 2, 3])
```

CHECKING THE TYPE

```
>>> type(a)
<type 'array'>
```

NUMERIC 'TYPE' OF ELEMENTS

```
>>> a.dtype
dtype('int32')
```

BYTES PER ELEMENT

```
>>> a.itemsize # per element
4
```

ARRAY SHAPE

```
# shape returns a tuple
# listing the length of the
# array along each dimension.
>>> a.shape
(4,)
>>> shape(a)
(4,)
```

ARRAY SIZE

```
# size reports the entire
# number of elements in an
# array.
>>> a.size
4
>>> size(a)
4
```

Introducing NumPy Arrays

BYTES OF MEMORY USED

```
# returns the number of bytes
# used by the data portion of
# the array.
 >>> a.nbytes
12
```

NUMBER OF DIMENSIONS

```
>>> a.ndim
1
```

ARRAY COPY

```
# create a copy of the array
>>> b = a.copy()
>>> b
array([0, 1, 2, 3])
```

CONVERSION TO LIST

```
# convert a numpy array to a
# python list.
>>> a.tolist()
[0, 1, 2, 3]

# For 1D arrays, list also
# works equivalently, but
# is slower.
>>> list(a)
[0, 1, 2, 3]
```

Setting Array Elements

ARRAY INDEXING

```
>>> a[0]
0
>>> a[0] = 10
>>> a
[10, 1, 2, 3]
```

FILL

```
# set all values in an array.
>>> a.fill(0)
>>> a
[0, 0, 0, 0]

# This also works, but may
# be slower.
>>> a[:] = 1
>>> a
[1, 1, 1, 1]
```

BEWARE OF TYPE COERSION

```
>>> a.dtype
dtype('int32')

# assigning a float to into # an int32
array will
# truncate decimal part.
>>> a[0] = 10.6
>>> a
[10, 1, 2, 3]

# fill has the same behavior
>>> a.fill(-4.8)
>>> a
[-4, -4, -4, -4]
```

Multi-Dimensional Arrays

MULTI-DIMENSIONAL ARRAYS

```
>>> a = array([[ 0, 1, 2, 3],
               [10,11,12,13]])
>>> a
array([[ 0, 1, 2, 3],
       [10,11,12,13]])
```

(ROWS,COLUMNS)

```
>>> a.shape
(2, 4)
>>> shape(a)
(2, 4)
```

ELEMENT COUNT

```
>>> a.size
8
>>> size(a)
8
```

NUMBER OF DIMENSIONS

```
>>> a.ndims
2
```

GET/SET ELEMENTS

```
>>> a[1,3]
13
```


```
>>> a[1,3] = -1
>>> a
array([[ 0, 1, 2, 3],
       [10,11,12,-1]])
```

ADDRESS FIRST ROW USING SINGLE INDEX

```
>>> a[1]
array([10, 11, 12, -1])
```

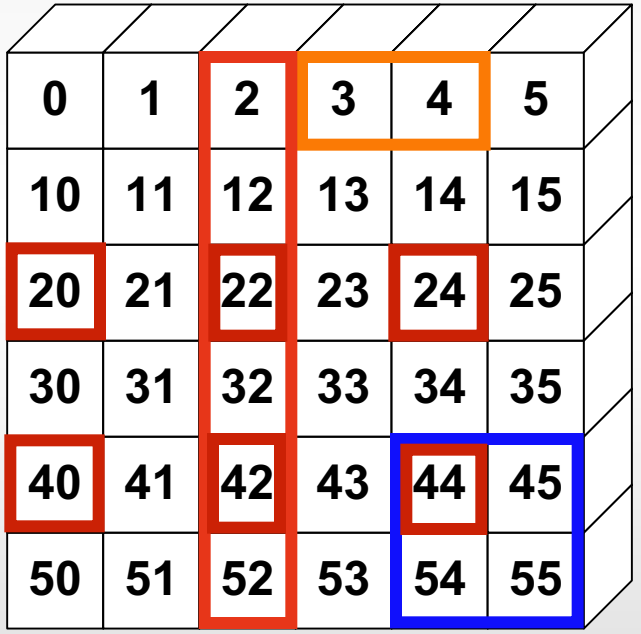
Array Slicing

SLICING WORKS MUCH LIKE STANDARD PYTHON SLICING

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

>>> a[4:,4:]
array([[44, 45],
       [54, 55]])

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

STRIDES ARE ALSO POSSIBLE

```
>>> a[2::2,::2]
array([[20, 22, 24],

       [40, 42, 44]])
```



Memory Model

```
>>> print a.strides
(24, 8)
>>> print a.flags.fortran, a.flags.contiguous
False True
>>> print a.T.strides
(8, 24)
>>> print a.T.flags.fortran, a.T.flags.contiguous
True False
```

Every dimension of an ndarray is accessed by stepping (striding) a fixed number of bytes through memory.

If memory is contiguous, then the strides are “pre-computed” indexing-formulas for either Fortran-order (first-dimension varies the fastest), or C-order (last-dimension varies the fastest) arrays.

Array slicing (Views)

Memory model allows “simple indexing” (integers and slices) into the array to be a **view** of the same data.

```
>>> b = a[:,::2]
>>> b[0,1] = 100
>>> print a
[[   1.    2.  100.]]
 [   4.    5.    6.]]
>>> c = a[:,::2].copy()
>>> c[1,0] = 500
>>> print a
[[   1.    2.  100.]]
 [   4.    5.    6.]]
```

Other uses of view

```
>>> b = a.view('i8')
>>> [hex(val.item()) for
val in b.flat]
['0x3FF0000000000000L',
 '0x4000000000000000L',
 '0x4059000000000000L',
 '0x4010000000000000L',
 '0x4014000000000000L',
 '0x4018000000000000L']
```

Slices are references to memory in original array. Changing values in a slice also changes the original array.

Slices Are References

- >>> a = array((0,1,2,3,4))

- # create a slice containing only the
- # last element of a
- >>> b = a[2:4]
- >>> b[0] = 10

- # changing b changed a!
- >>> a
- array([1, 2, 10, 3, 4])

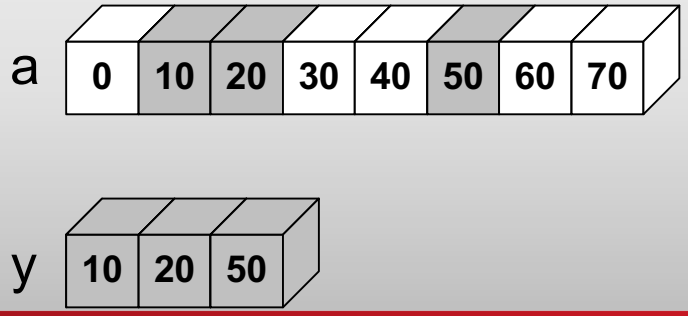
Fancy Indexing

INDEXING BY POSITION

```
>>> a = arange(0,80,10)


# fancy indexing
>>> y = a[[1, 2, -3]]
>>> print y
[10 20 50]

# using take
>>> y = take(a,[1,2,-3])
>>> print y
[10 20 50]
```

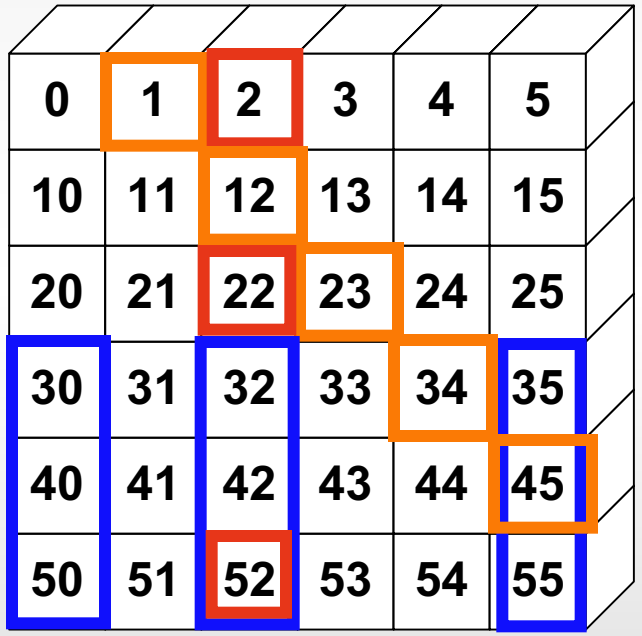
INDEXING WITH BOOLEANS

```
>>> mask = array([0,1,1,0,0,1,0,0],
...              dtype=bool)

# fancy indexing
>>> y = a[mask]
>>> print y
[10,20,50]

# using compress
>>> y = compress(mask, a)
>>> print y
[10,20,50]
```



Fancy Indexing in 2D

```
>>> a[(0,1,2,3,4),(1,2,3,4,5)]
array([ 1, 12, 23, 34, 45])

>>> a[3:,[0, 2, 5]]
array([[30, 32, 35],
       [40, 42, 45]])
       [50, 52, 55]])

>>> mask = array([1,0,1,0,0,1],
                 dtype=bool)
>>> a[mask,2]
array([2,22,52])
```



Unlike slicing, fancy indexing creates copies instead of views into original arrays.

Data-types

There are two related concepts of “type”

- The data-type object (dtype)
- The Python “type” of the object created from a single array item (hierarchy of scalar types)

The **dtype** object provides the details of how to interpret the memory for an item. It's an instance of a single dtype class.

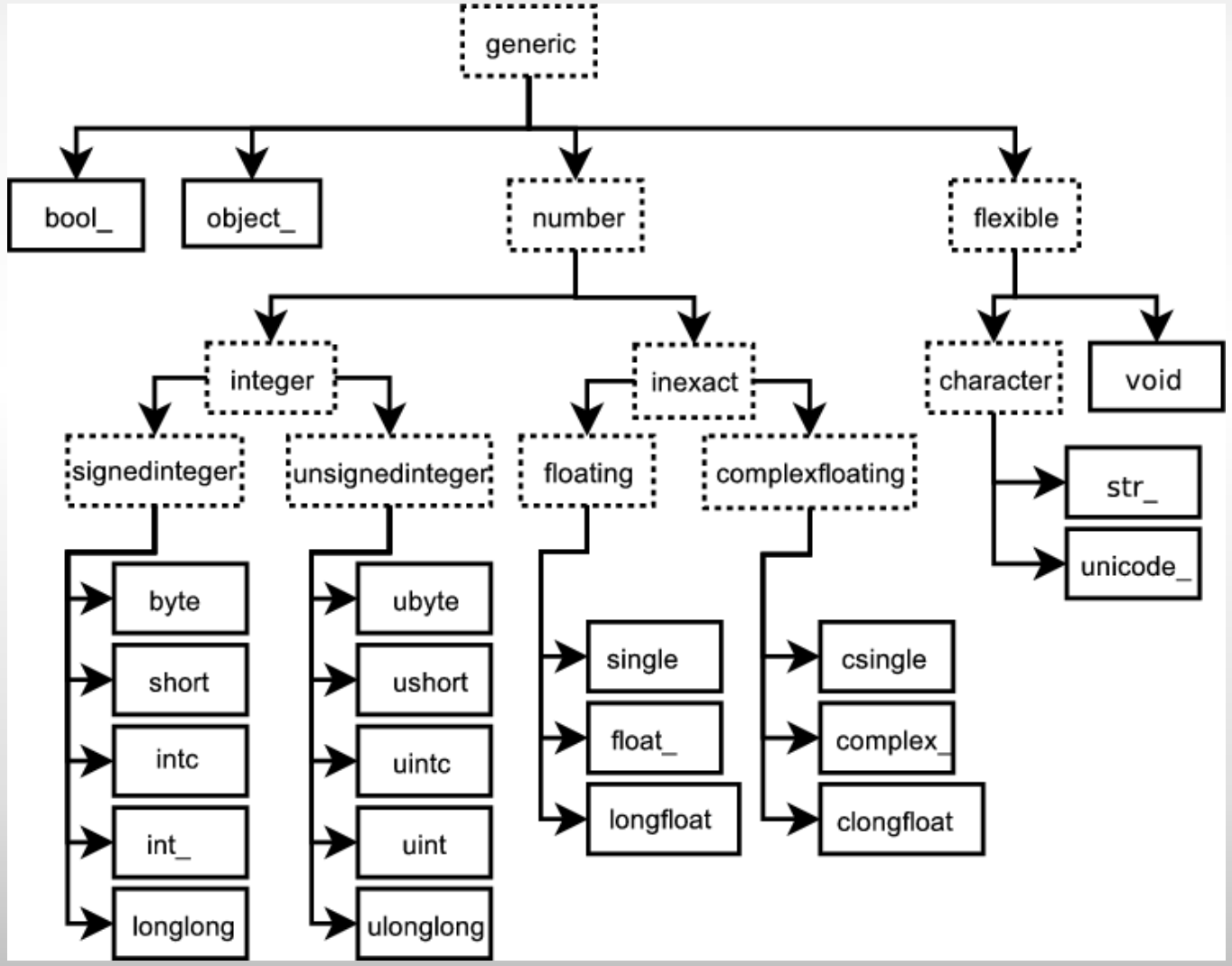
The “type” of the extracted elements are true Python classes that exist in a hierarchy of Python classes

Every dtype object has a type attribute which provides the Python object returned when an element is selected from the array

NumPy dtypes

Basic Type	Available NumPy types	Comments
Boolean	bool	Elements are 1 byte in size
Integer	int8, int16, int32, int64, int128, int	int defaults to the size of int in C for the platform
Unsigned Integer	uint8, uint16, uint32, uint64, uint128, uint	uint defaults to the size of unsigned int in C for the platform
Float	float32, float64, float, longfloat,	Float is always a double precision floating point value (64 bits). longfloat represents large precision floats. Its size is platform dependent.
Complex	complex64, complex128, complex	The real and complex elements of a complex64 are each represented by a single precision (32 bit) value for a total size of 64 bits.
Strings	str, unicode	Unicode is always UTF32 (UCS4)
Object	object	Represent items in array as Python objects.
Records	void	Used for arbitrary data structures in record arrays.

Built-in “scalar” types



Data-type object (dtype)

There are 21 “built-in” (static) data-type objects

New (dynamic) data-type objects are created to handle

- Alteration of the byteorder
- Change in the element size (for string, unicode, and void built-ins)
- Addition of fields
- Change of the type object (C-structure arrays)

Creation of data-types is quite flexible.

New user-defined “built-in” data-types can also be added (but must be done in C and involves filling a function-pointer table)

Data-type fields

An item can include fields of different data-types.

A field is described by a data-type object and a byte offset -- - this definition allows nested records.

The array construction command interprets tuple elements as field entries.

```
>>> dt = N.dtype(“i4,f8,a5”)

>>> print dt.fields
{'f1': (dtype('<i4'), 0), 'f2': (dtype('<f8'), 4), 'f3':
(dtype('|S5'), 12)}

>>> a = N.array([(1,2.0,”Hello”), (2,3.0,”World”)], dtype=dt)

>>> print a[’f2']
[Hello World]
```

Array Calculation Methods

SUM FUNCTION

```
>>> a = array([[1,2,3],
               [4,5,6]], float)

# Sum defaults to summing all
# *all* array values.
>>> sum(a)
21.

# supply the keyword axis to
# sum along the 0th axis.
>>> sum(a, axis=0)
array([5., 7., 9.])

# supply the keyword axis to
# sum along the last axis.
>>> sum(a, axis=-1)
array([6., 15.])
```

SUM ARRAY METHOD

```
# The a.sum() defaults to
# summing *all* array values
>>> a.sum()
21.

# Supply an axis argument to
# sum along a specific axis.
>>> a.sum(axis=0)
array([5., 7., 9.])
```

PRODUCT

```
# product along columns.
>>> a.prod(axis=0)
array([ 4., 10., 18.])

# functional form.
>>> prod(a, axis=0)
array([ 4., 10., 18.])
```

Min/Max

MIN

```
>>> a = array([2.,3.,0.,1.])
>>> a.min(axis=0)
0.
# use Numpy's amin() instead
# of Python's builtin min()
# for speed operations on
# multi-dimensional arrays.
>>> amin(a, axis=0)
0.
```

MAX

```
>>> a = array([2.,1.,0.,3.]) >>>
a.max(axis=0)
3.

# functional form
>>> amax(a, axis=0)
3.
```

ARGMIN

```
# Find index of minimum value.
>>> a.argmin(axis=0)
2
# functional form
>>> argmin(a, axis=0)
2
```

ARGMAX

```
# Find index of maximum value.
>>> a.argmax(axis=0)
1
# functional form
>>> argmax(a, axis=0)
1
```

Statistics Array Methods

MEAN

```
>>> a = array([[1,2,3],
              [4,5,6]], float)

# mean value of each column
>>> a.mean(axis=0)
array([ 2.5,  3.5,  4.5])
>>> mean(a, axis=0)
array([ 2.5,  3.5,  4.5])
>>> average(a, axis=0)
array([ 2.5,  3.5,  4.5])

# average can also calculate
# a weighted average
>>> average(a, weights=[1,2],
...         axis=0)
array([ 3., 4., 5.])
```

STANDARD DEV./VARIANCE

```
# Standard Deviation
>>> a.std(axis=0)
array([ 1.5,  1.5,  1.5])

# Variance
>>> a.var(axis=0)
array([2.25, 2.25, 2.25])
>>> var(a, axis=0)
array([2.25, 2.25, 2.25])
```

Other Array Methods

CLIP

```
# Limit values to a range

>>> a = array([[1,2,3],
               [4,5,6]], float)

# Set values < 3 equal to 3.
# Set values > 5 equal to 5.
>>> a.clip(3,5)
>>> a
array([[ 3.,  3.,  3.],
       [ 4.,  5.,  5.]])
```

ROUND

```
# Round values in an array.
# Numpy rounds to even, so
# 1.5 and 2.5 both round to 2.
>>> a = array([1.35, 2.5, 1.5])
>>> a.round()
array([ 1.,  2.,  2.])

# Round to first decimal place.
>>> a.round(decimals=1)
array([ 1.4,  2.5,  1.5])
```

POINT TO POINT

```
# Calculate max – min for
# array along columns
>>> a.ptp(axis=0)
array([ 3.0,  3.0,  3.0])
# max – min for entire array.
>>> a.ptp(axis=None)
5.0
```

Array attributes/methods

BASIC ATTRIBUTES

```
a.dtype – Numerical type of array elements. float32, uint8, etc.
a.shape – Shape of the array. (m,n,o,...)
a.size – Number of elements in entire array.
a.itemsize – Number of bytes used by a single element in the array.
a.nbytes – Number of bytes used by entire array (data only).
a.ndim – Number of dimensions in the array.
```

SHAPE OPERATIONS

```
a.flat – An iterator to step through array as if it is 1D.
a.flatten() – Returns a 1D copy of a multi-dimensional array.
a.ravel() – Same as flatten(), but returns a 'view' if possible.
a.resize(new_size) – Change the size/shape of an array in-place.
a.swapaxes(axis1, axis2) – Swap the order of two axes in an array.
a.transpose(*axes) – Swap the order of any number of array axes.
a.T – Shorthand for a.transpose()
a.squeeze() – Remove any length=1 dimensions from an array.
```

Array attributes/methods

FILL AND COPY

```
a.copy() – Return a copy of the array.
a.fill(value) – Fill array with a scalar value.
```

CONVERSION / COERSION

```
a.tolist() – Convert array into nested lists of values.
a.tostring() – raw copy of array memory into a python string.
a.astype(dtype) – Return array coerced to given dtype.
a.byteswap(False) – Convert byte order (big <-> little endian).
```

COMPLEX NUMBERS

```
a.real – Return the real part of the array.
a.imag – Return the imaginary part of the array.
a.conjugate() – Return the complex conjugate of the array.
a.conj()– Return the complex conjugate of an array.(same as conjugate)
```

Array attributes/methods

SAVING

```
a.dump(file) - Store a binary array data out to the given file.
a.dumps() - returns the binary pickle of the array as a string.
a.tofile(fid, sep="", format="%s") Formatted ascii output to file.
```

SEARCH / SORT

```
a.nonzero() - Return indices for all non-zero elements in a.
a.sort(axis=-1) - Inplace sort of array elements along axis.
a.argsort(axis=-1) - Return indices for element sort order along axis.
a.searchsorted(b) - Return index where elements from b would go in a.
```

ELEMENT MATH OPERATIONS

```
a.clip(low, high) - Limit values in array to the specified range.
a.round(decimals=0) - Round to the specified number of digits.
a.cumsum(axis=None) - Cumulative sum of elements along axis.
a.cumprod(axis=None) - Cumulative product of elements along axis.
```

Array attributes/methods

REDUCTION METHODS

All the following methods "reduce" the size of the array by 1 dimension by carrying out an operation along the specified axis. If axis is None, the operation is carried out across the entire array.

```
a.sum(axis=None) – Sum up values along axis.
a.prod(axis=None) – Find the product of all values along axis.
a.min(axis=None)– Find the minimum value along axis.
a.max(axis=None) – Find the maximum value along axis.
a.argmin(axis=None) – Find the index of the minimum value along axis.
a.argmax(axis=None) – Find the index of the maximum value along axis.
a.ptp(axis=None) – Calculate a.max(axis) – a.min(axis)
a.mean(axis=None) – Find the mean (average) value along axis.
a.std(axis=None) – Find the standard deviation along axis.
a.var(axis=None) – Find the variance along axis.

a.any(axis=None) – True if any value along axis is non-zero.   (or)
a.all(axis=None) – True if all values along axis are non-zero. (and)
```

Array Operations

SIMPLE ARRAY MATH

```
>>> a = array([1,2,3,4])
>>> b = array([2,3,4,5])
>>> a + b
array([3, 5, 7, 9])
```

Numpy defines the following constants:

pi = 3.14159265359

e = 2.71828182846

MATH FUNCTIONS

```
# Create array from 0 to 10
>>> x = arange(11.)

# multiply entire array by
# scalar value
>>> a = (2*pi)/10.
>>> a
0.6283185307179586
>>> a*x
array([ 0.,0.628,…,6.283])

# inplace operations
>>> x *= a
>>> x
array([ 0.,0.628,…,6.283])

# apply functions to array.
>>> y = sin(x)
```

Universal Functions

ufuncs are objects that rapidly evaluate a function element-by-element over an array.

Core piece is a 1-d loop written in C that performs the operation over the largest dimension of the array

For 1-d arrays it is equivalent to but much faster than list comprehension

```
>>> type(N.exp)
<type 'numpy.ufunc'>
>>> x = array([1,2,3,4,5])
>>> print N.exp(x)
[   2.71828183    7.3890561    20.08553692
54.59815003  148.4131591 ]
>>> print [math.exp(val) for val in x]
[2.7182818284590451,
7.3890560989306504,20.085536923187668,
 54.598150033144236,148.4131591025766]
```

Mathematic Binary Operators

a + b → add(a,b)
a - b → subtract(a,b)
a % b → remainder(a,b)

a * b → multiply(a,b)
a / b → divide(a,b)
a ** b → power(a,b)

MULTIPLY BY A SCALAR

```
>>> a = array((1,2))
>>> a*3.
array([3., 6.])
```

ELEMENT BY ELEMENT ADDITION

```
>>> a = array([1,2])
>>> b = array([3,4])
>>> a + b
array([4, 6])
```

ADDITION USING AN OPERATOR FUNCTION

```
>>> add(a,b)
array([4, 6])
```

IN PLACE OPERATION

```
# Overwrite contents of a.
# Saves array creation
# overhead
>>> add(a,b,a) # a += b
array([4, 6])
>>> a
array([4, 6])
```

Comparison and Logical Operators

```
equal          (==)
greater_equal  (>=)
logical_and
logical_not
```

```
not_equal   (!=)
less        (<)
logical_or
```

```
greater     (>)
less_equal  (<=)
logical_xor
```

2D EXAMPLE

```
>>> a = array(((1,2,3,4),(2,3,4,5)))
>>> b = array(((1,2,5,4),(1,3,4,5)))
>>> a == b
array([[True, True, False, True], [False, True, True, True]])
# functional equivalent
>>> equal(a,b)
array([[True, True, False, True], [False, True, True, True]])
```

Bitwise Operators

bitwise_and (&)	invert (~)	right_shift(a,shifts)
bitwise_or (|)	bitwise_xor	left_shift (a,shifts)

BITWISE EXAMPLES

```
>>> a = array((1,2,4,8))
>>> b = array((16,32,64,128))
>>> bitwise_or(a,b)
array([ 17,  34,  68, 136])

# bit inversion
>>> a = array((1,2,3,4), uint8)
>>> invert(a)
array([254, 253, 252, 251], dtype=uint8)

# left shift operation
>>> left_shift(a,3)
array([ 8, 16, 24, 32], dtype=uint8)
```

Trig and Other Functions

TRIGONOMETRIC

```
sin(x)        sinh(x)
cos(x)        cosh(x)
arccos(x)     arccosh(x)

arctan(x)     arctanh(x)
arcsin(x)     arcsinh(x)
arctan2(x,y)
```

OTHERS

```
exp(x)        log(x)
log10(x)      sqrt(x)
absolute(x)   conjugate(x)
negative(x)   ceil(x)
floor(x)      fabs(x)
hypot(x,y)    fmod(x,y)
maximum(x,y)  minimum(x,y)
```

hypot(x,y)

Element by element distance calculation using $\sqrt{x^2 + y^2}$

Broadcasting

When there are multiple inputs, then they all must be "broadcastable" to the same shape.

- All arrays are promoted to the same number of dimensions (by pre-prending 1's to the shape)
- All dimensions of length 1 are expanded as determined by other inputs with non-unit lengths in that dimension.

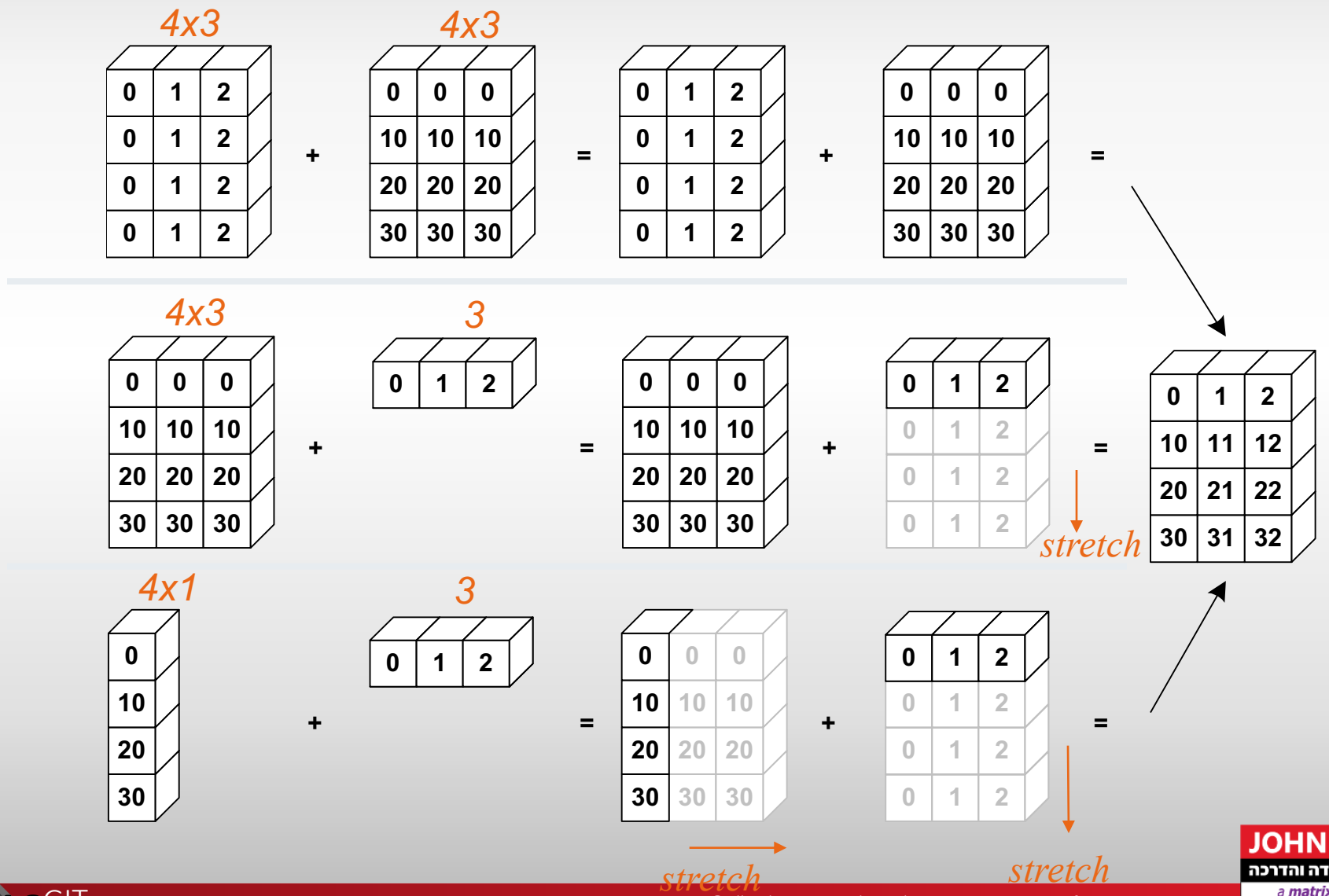
```
>>> x = [1,2,3,4];
>>> y =
[[10],[20],[30]]
>>> print N.add(x,y)
[[11 12 13 14]
 [21 22 23 24]
 [31 32 33 34]]
>>> x = array(x)
>>> y = array(y)
>>> print x+y
[[11 12 13 14]
 [21 22 23 24]
 [31 32 33 34]]
```

x has shape (4,) the ufunc sees it as having shape (1,4)

y has shape (3,1)

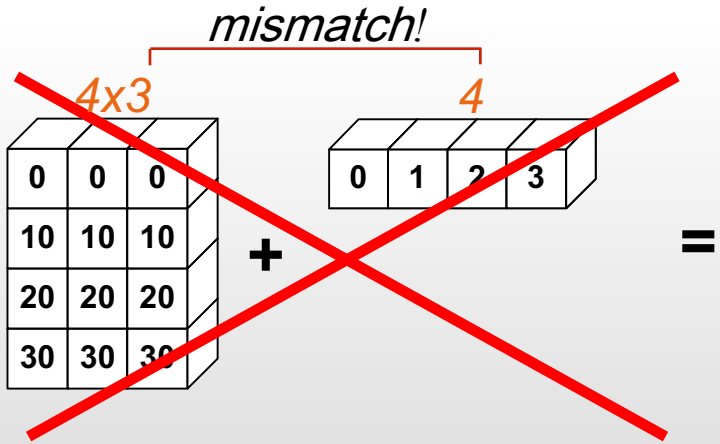
The ufunc result has shape (3,4)

Array Broadcasting



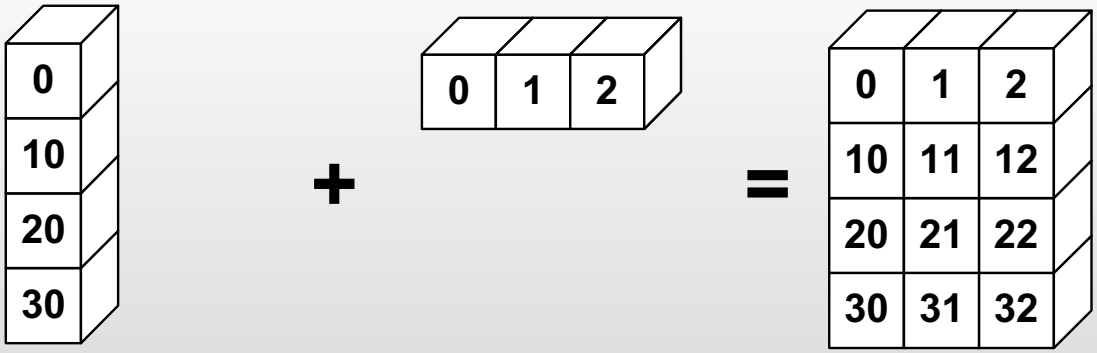
Broadcasting Rules

The *trailing* axes of both arrays must either be 1 or have the same size for broadcasting to occur. Otherwise, a "`ValueError: frames are not aligned`" exception is thrown.



Broadcasting in Action

```
>>> a = array((0,10,20,30))
>>> b = array((0,1,2))
>>> y = a[:, None] + b
```



Universal Function Methods

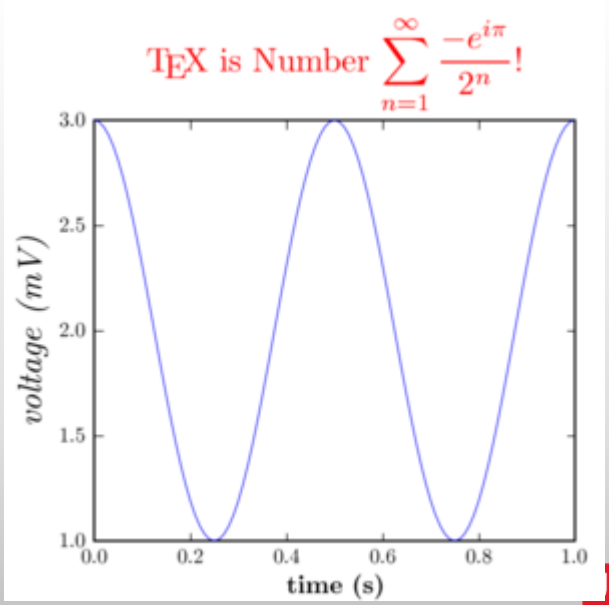
The mathematic, comparative, logical, and bitwise operators that take two arguments (binary operators) have special methods that operate on arrays:

```
op.reduce(a,axis=0)
op.accumulate(a,axis=0)
op.outer(a,b)
op.reduceat(a,indices)
```

Matplotlib

Requires NumPy extension. Provides powerful plotting commands.

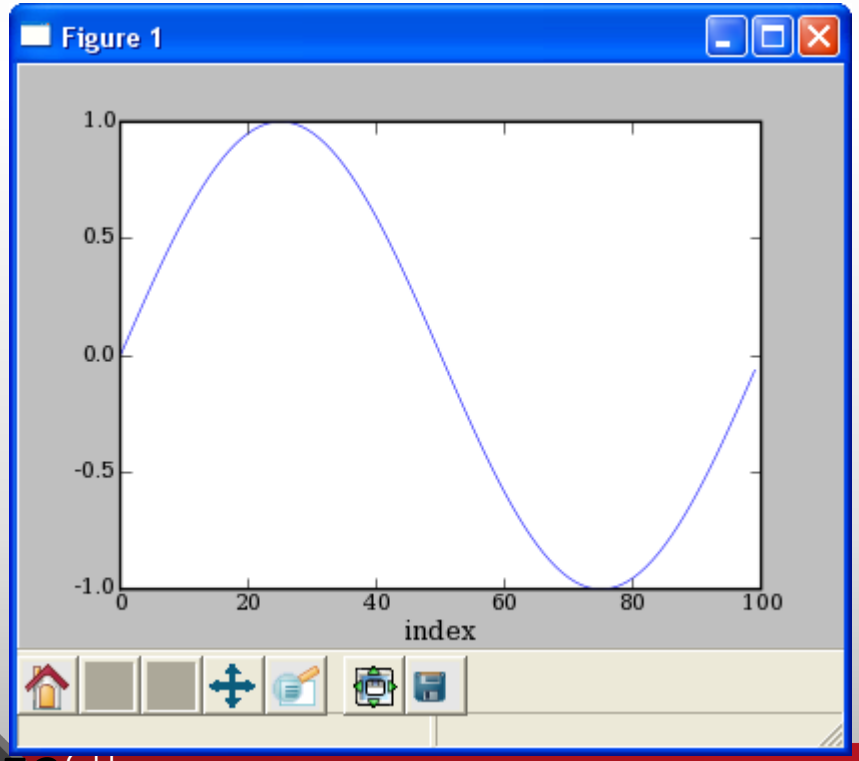
http://matplotlib.sourceforge.net



Line Plots

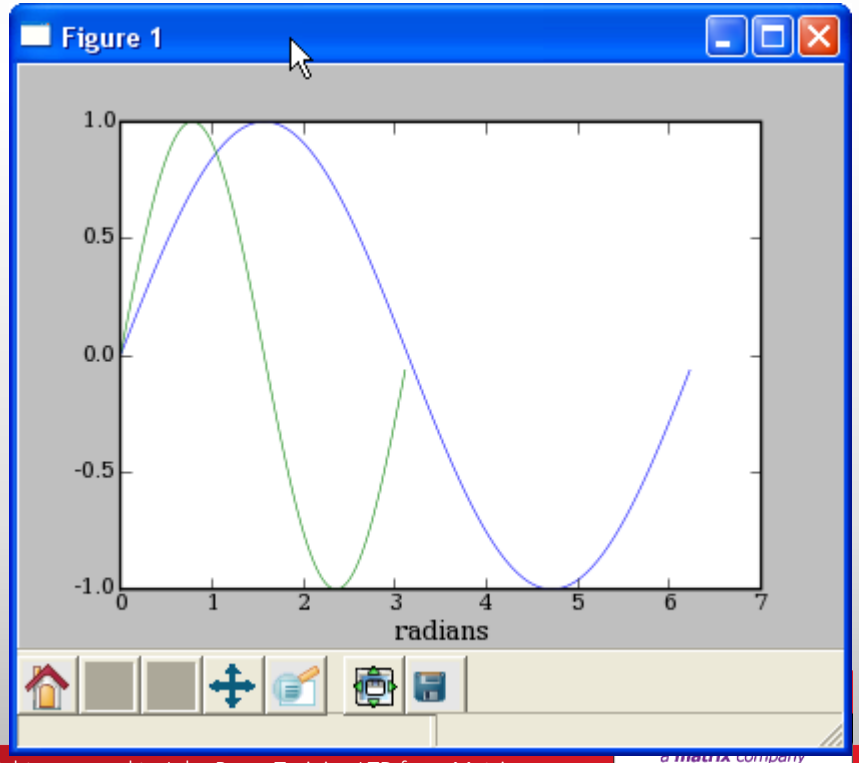
PLOT AGAINST INDICES

```
>>> x = arange(50)*2*pi/50.
>>> y = sin(x)
>>> plot(y)
>>> xlabel('index')
```



MULTIPLE DATA SETS

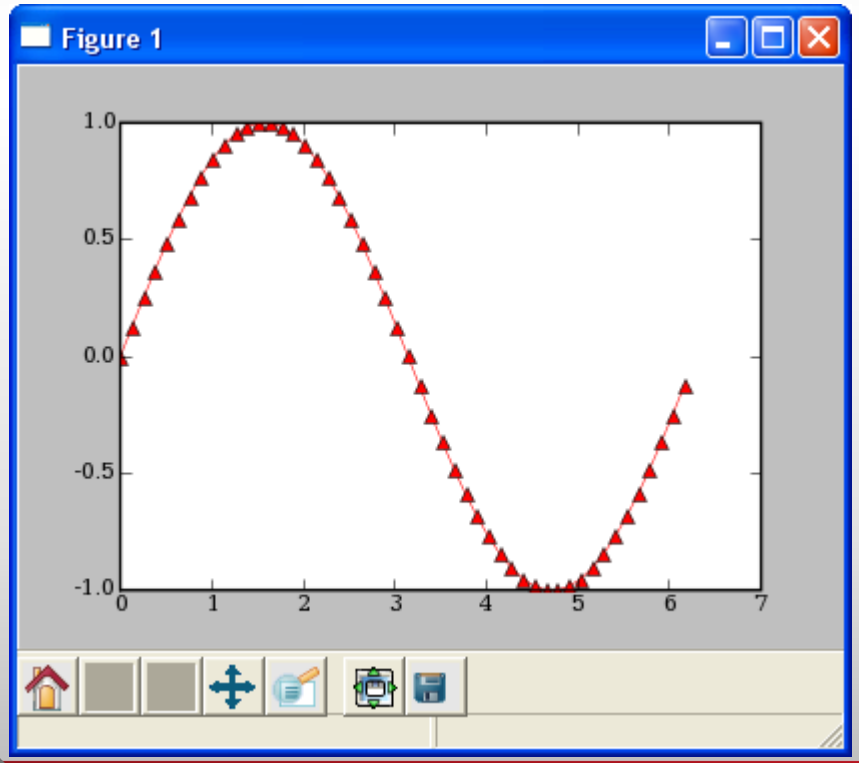
```
>>> plot(x,y,x2,y2)
>>> xlabel('radians')
```



Line Plots

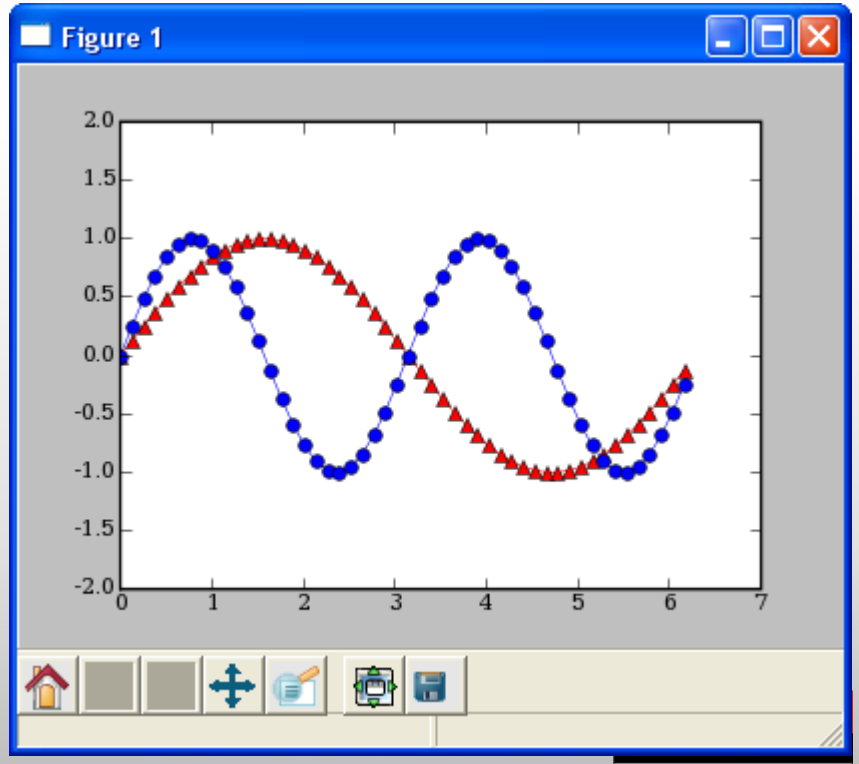
LINE FORMATTING

```
# red, dot-dash, triangles
>>> plot(x,sin(x),'r-^')
```



MULTIPLE PLOT GROUPS

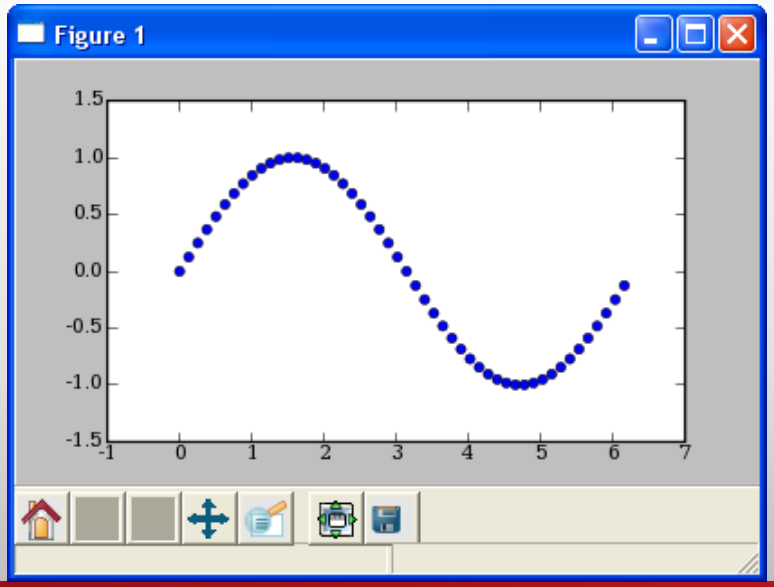
```
>>> plot(x,y1,'b-o', x,y2), r-^')
>>> axis([0,7,-2,2])
```



Scatter Plots

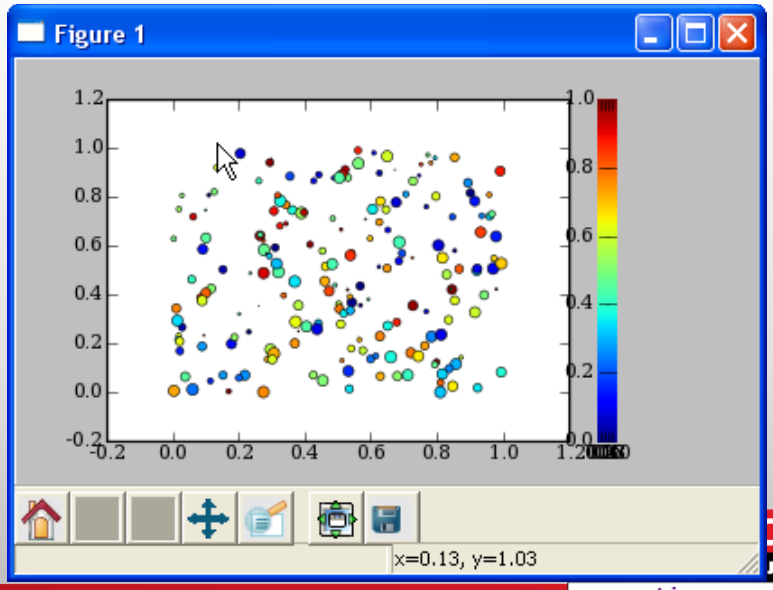
SIMPLE SCATTER PLOT

```
>>> x = arange(50)*2*pi/50.
>>> y = sin(x)
>>> scatter(x,y)
```



COLORMAPPED SCATTER

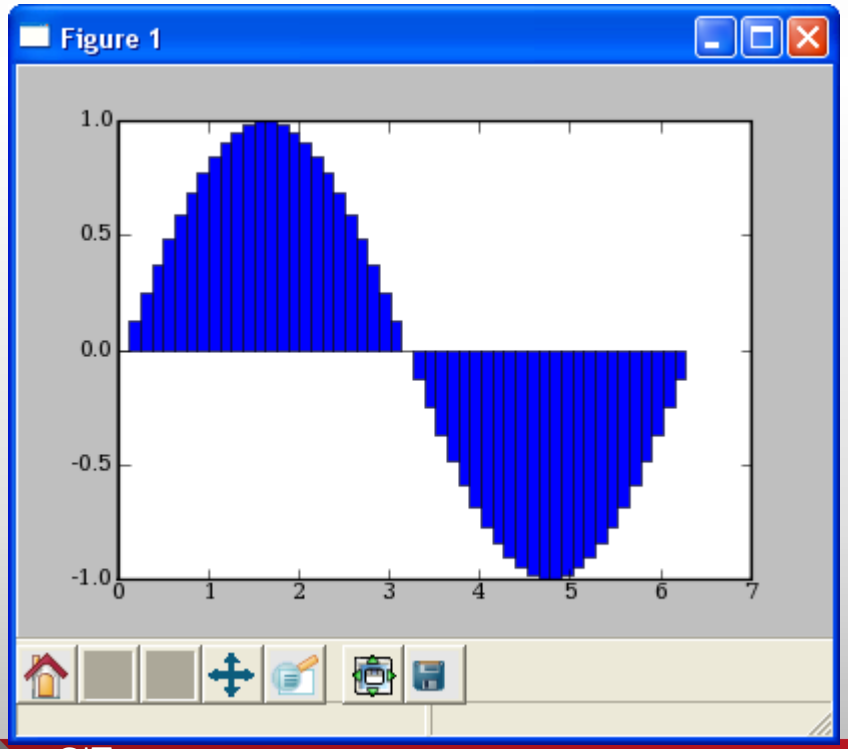
```
# marker size/color set with data
>>> x = rand(200)
>>> y = rand(200)
>>> size = rand(200)*30
>>> color = rand(200)
>>> scatter(x, y, size, color)
>>> colorbar()
```



Bar Plots

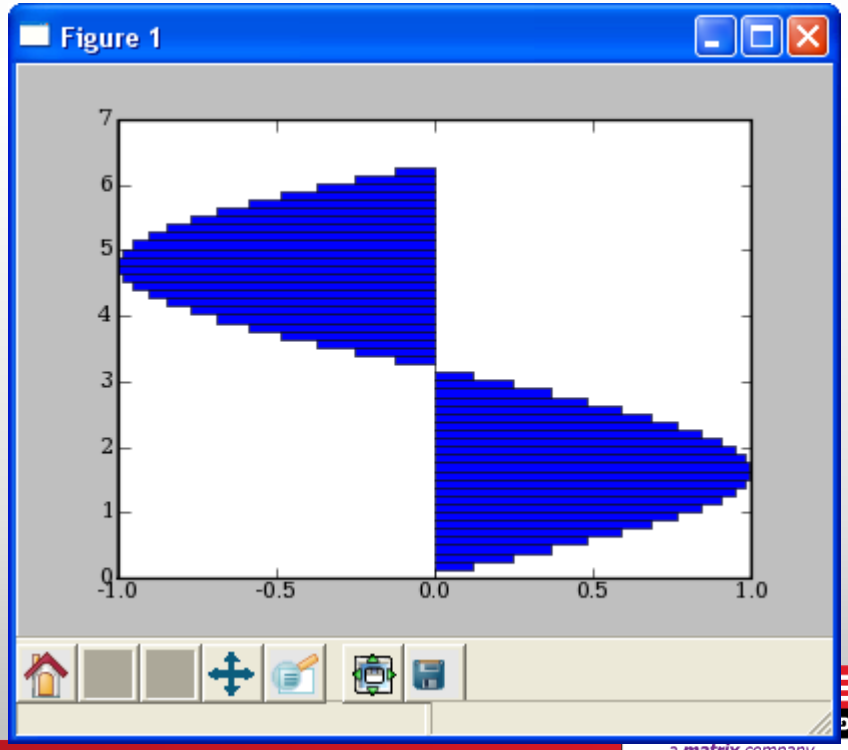
BAR PLOT

```
>>> bar(x,sin(x),
...     width=x[1]-x[0])
```



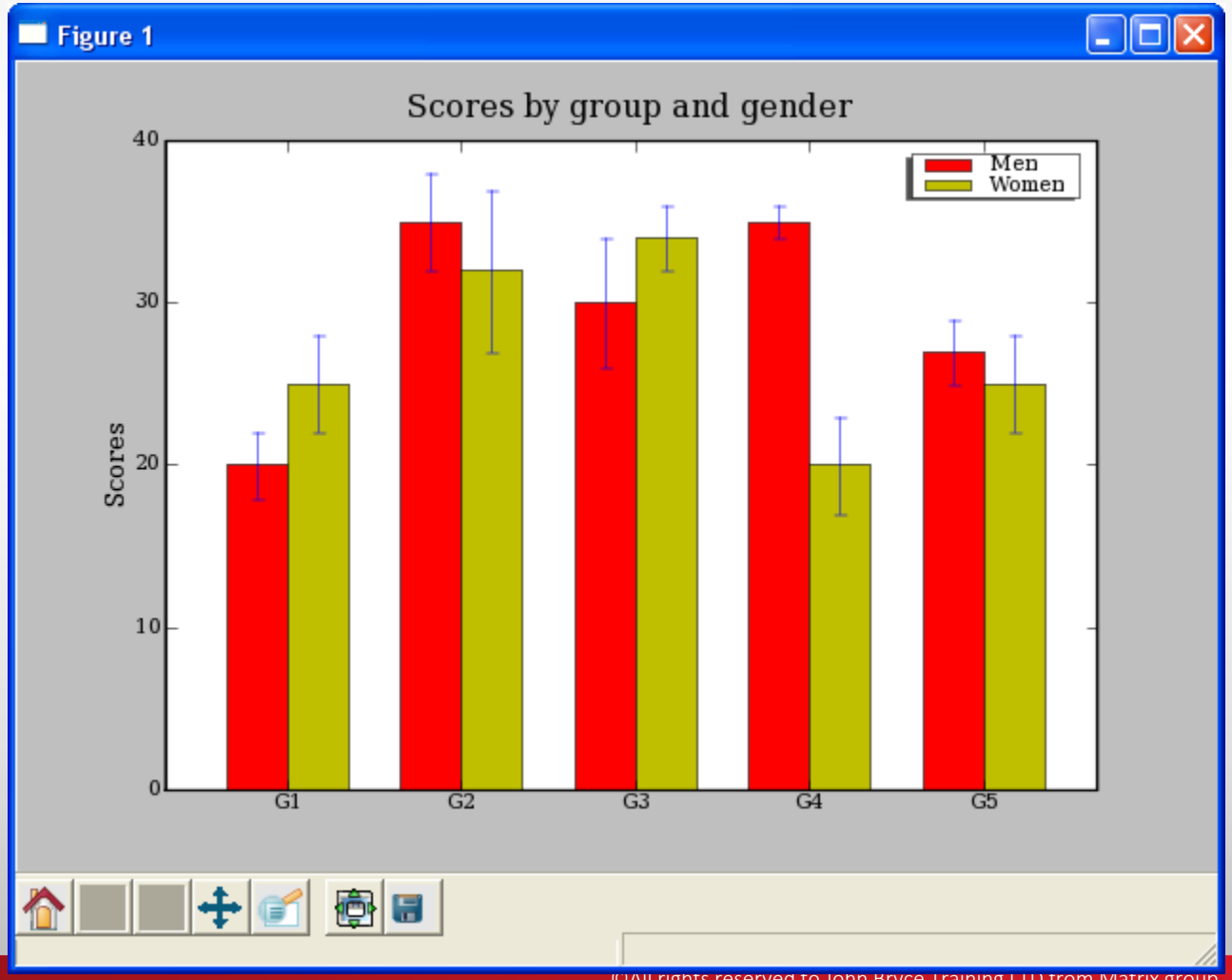
HORIZONTAL BAR PLOT

```
>>> barh(x,sin(x),
...     height=x[1]-x[0])
```



Bar Plots

DEMO/MATPLOTLIB_PLOTTING/BARCHART_DEMO.PY

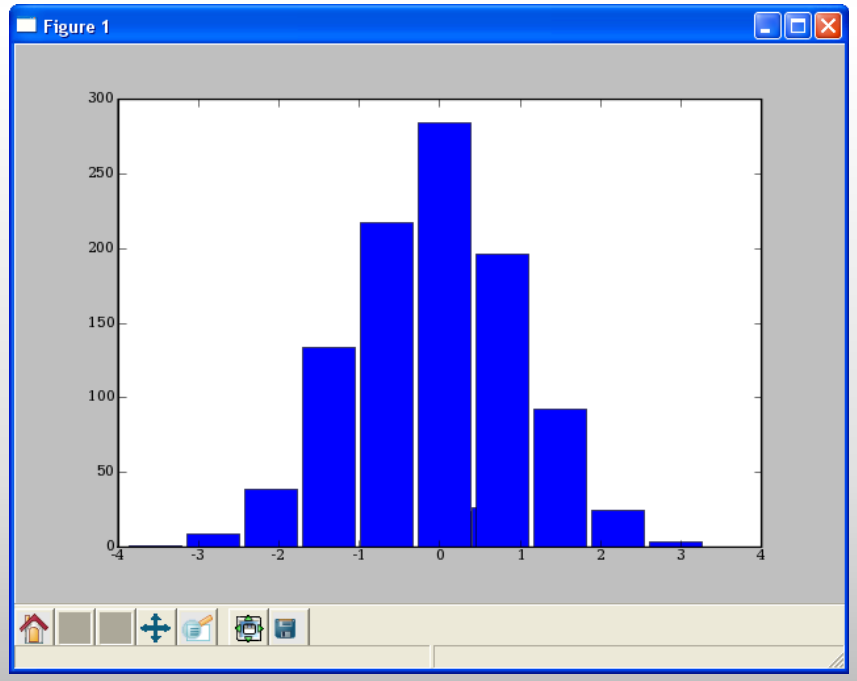


```python
import numpy as np
import matplotlib.pyplot as plt

N = 5
menMeans = (20, 35, 30, 35, 27)
menStd =   (2, 3, 4, 1, 2)

ind = np.arange(N)  # the x locations for the groups
width = 0.35       # the width of the bars

fig, ax = plt.subplots()
rects1 = ax.bar(ind, menMeans, width, color='r', yerr=menStd)

womenMeans = (25, 32, 34, 20, 25)
womenStd =   (3, 5, 2, 3, 3)
rects2 = ax.bar(ind+width, womenMeans, width, color='y', yerr=womenStd)

# add some text for labels, title and axes ticks
ax.set_ylabel('Scores')
ax.set_title('Scores by group and gender')
ax.set_xticks(ind+width)
ax.set_xticklabels( ('G1', 'G2', 'G3', 'G4', 'G5') )

ax.legend( (rects1[0], rects2[0]), ('Men', 'Women') )

def autolabel(rects):
    # attach some text labels
    for rect in rects:
        height = rect.get_height()
        ax.text(rect.get_x()+rect.get_width()/2., 1.05*height, '%d'%int(height),
                ha='center', va='bottom')

autolabel(rects1)
autolabel(rects2)

plt.show()
```

HISTOGRAMS

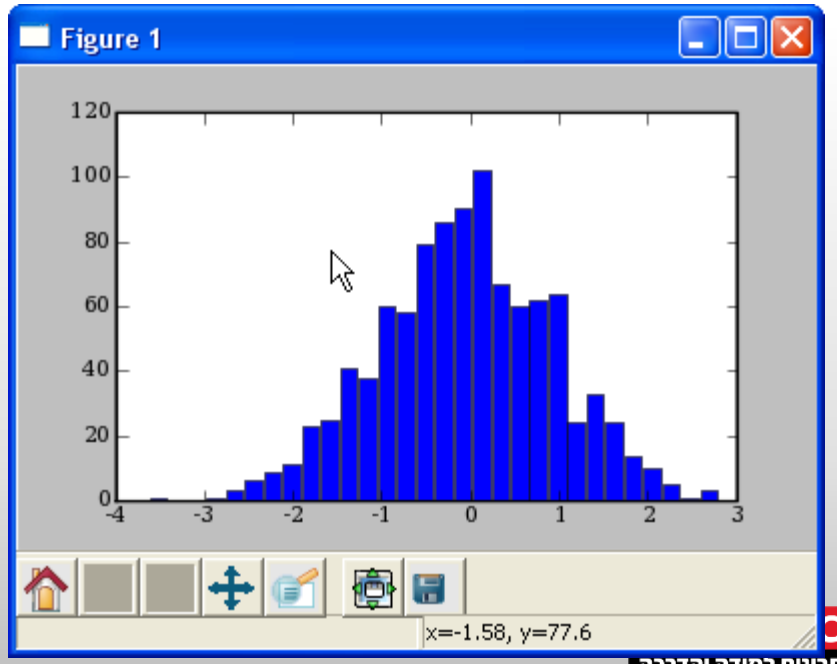
HISTOGRAM

```
# plot histogram
# default to 10 bins
>>> hist(randn(1000))
```



HISTOGRAM 2

```
# change the number of bins
>>> hist(randn(1000), 30)
```



Multiple Plots using Subplot

DEMO/MATPLOTLIB_PLOTTING/EXAMPLES/SUBPLOT_DEMO.PY

```
def f(t):
    s1 = cos(2*pi*t)
    e1 = exp(-t)
    return multiply(s1,e1)

t1 = arange(0.0, 5.0, 0.1)
t2 = arange(0.0, 5.0, 0.02)
t3 = arange(0.0, 2.0, 0.01)

subplot(211)
l = plot(t1, f(t1), 'bo', t2, f(t2),
        'k--')
setp(l, 'markerfacecolor', 'g')
grid(True)
title('A tale of 2 subplots')
ylabel('Damped oscillation')

subplot(212)
plot(t3, cos(2*pi*t3), 'r.')
grid(True)
xlabel('time (s)')
ylabel('Undamped')
show()
```

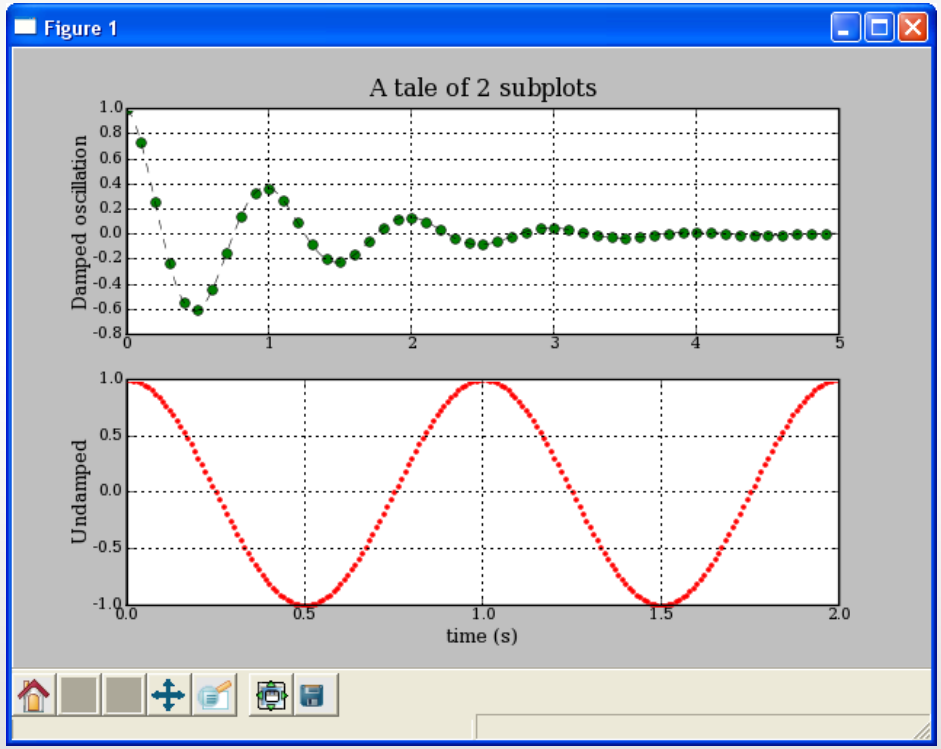
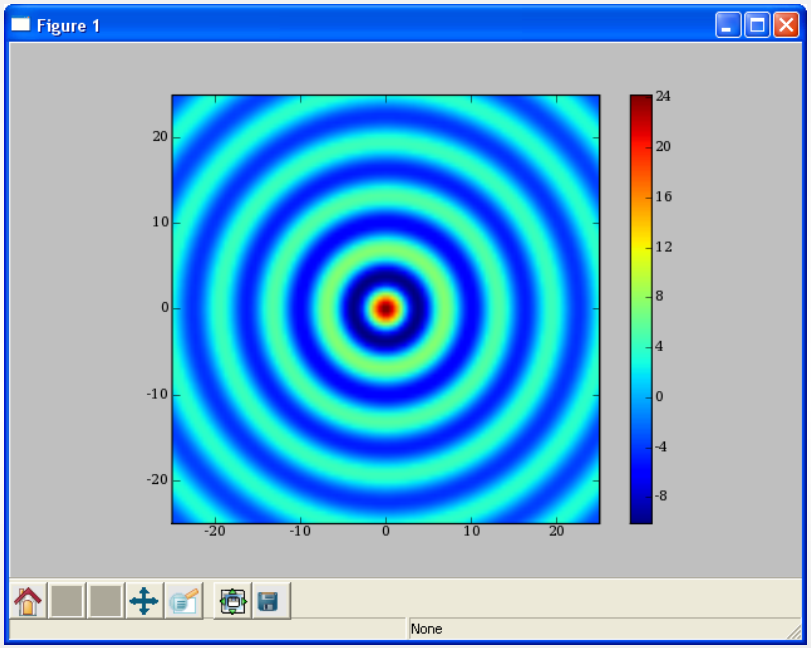


Image demo

```
import matplotlib.pyplot as plt
import matplotlib.cbook as cbook

image_file = cbook.get_sample_data('/Users/liran/hires.png')
image = plt.imread(image_file)

plt.imshow(image)
plt.axis('off') # clear x- and y-axes
plt.show()
```

Image Display

```
# Create 2d array where values
# are radial distance from
# the center of array.
>>> from numpy import mgrid
>>> from scipy import special
>>> x,y = mgrid[-25:25:100j,
...             -25:25:100j]
>>> r = sqrt(x**2+y**2)
# Calculate bessel function of
# each point in array and scale
>>> s = special.j0(r)*25

# Display surface plot.
>>> imshow(s, extent=[-25,25,-25,25])
>>> colorbar()
```



Animation

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

def update_line(num, data, line):
    line.set_data(data[...,:num])
    return line,

fig1 = plt.figure()

data = np.random.rand(2, 25)
l, = plt.plot([], [], 'r-')
plt.xlim(0, 1)
plt.ylim(0, 1)
plt.xlabel('x')
plt.title('test')
line_ani = animation.FuncAnimation(fig1, update_line, 25, fargs=(data, l),
    interval=50, blit=True)
#line_ani.save('lines.mp4')

fig2 = plt.figure()

x = np.arange(-9, 10)
y = np.arange(-9, 10).reshape(-1, 1)
base = np.hypot(x, y)
ims = []
for add in np.arange(15):
    ims.append((plt.pcolor(x, y, base + add, norm=plt.Normalize(0, 30)),))

im_ani = animation.ArtistAnimation(fig2, ims, interval=50, repeat_delay=3000,
    blit=True)
#im_ani.save('/Users/liran/im.mp4', metadata={'artist':'Guido'})

plt.show()
```

Events

```
from __future__ import print_function
import matplotlib.pyplot as plt

def handle_close(evt):
    print('Closed Figure!')

fig = plt.figure()
fig.canvas.mpl_connect('close_event', handle_close)

plt.text(0.35, 0.5, 'Close Me!', dict(size=30))
plt.show()
```

Saving figures

Matplotlib can generate high-quality output in a number formats, including PNG, JPG, EPS, SVG, PGF and PDF.

fig.savefig("filename.png")

fig.savefig("filename.png”, dpi=200)

SciPy Overview

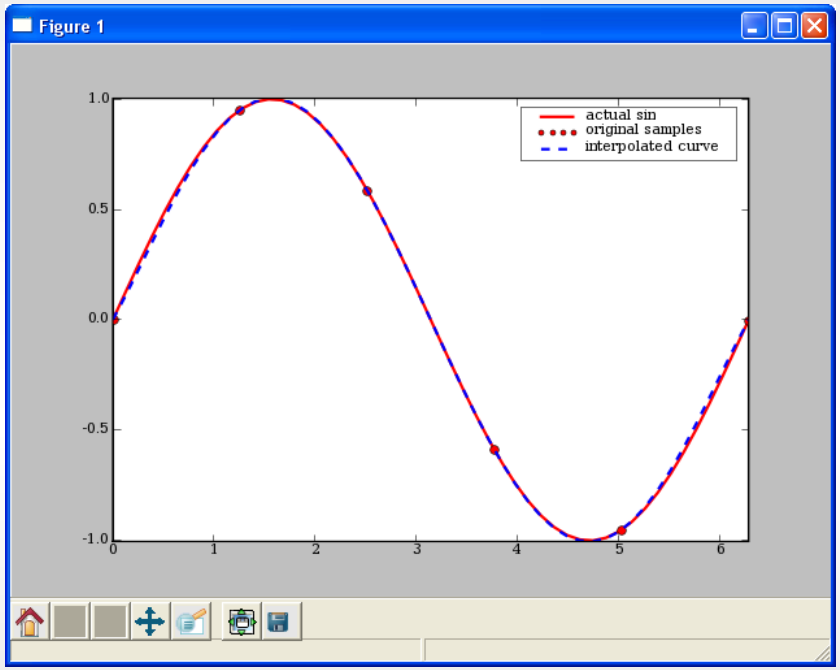
- Available at www.scipy.org
- Open Source BSD Style License
- Over 30 svn "committers" to the project

CURRENT PACKAGES

- Special Functions (scipy.special)
- Signal Processing (scipy.signal)
- Image Processing (scipy.ndimage)
- Fourier Transforms (scipy.fftpack)
- Optimization (scipy.optimize)
- Numerical Integration (scipy.integrate)
- Linear Algebra (scipy.linalg)
- Input/Output (scipy.io)
- Statistics (scipy.stats)
- Fast Execution (scipy.weave)
- Clustering Algorithms (scipy.cluster)
- Sparse Matrices (scipy.sparse)
- Interpolation (scipy.interpolate)
- More (e.g. scipy.odr, scipy.maxentropy)

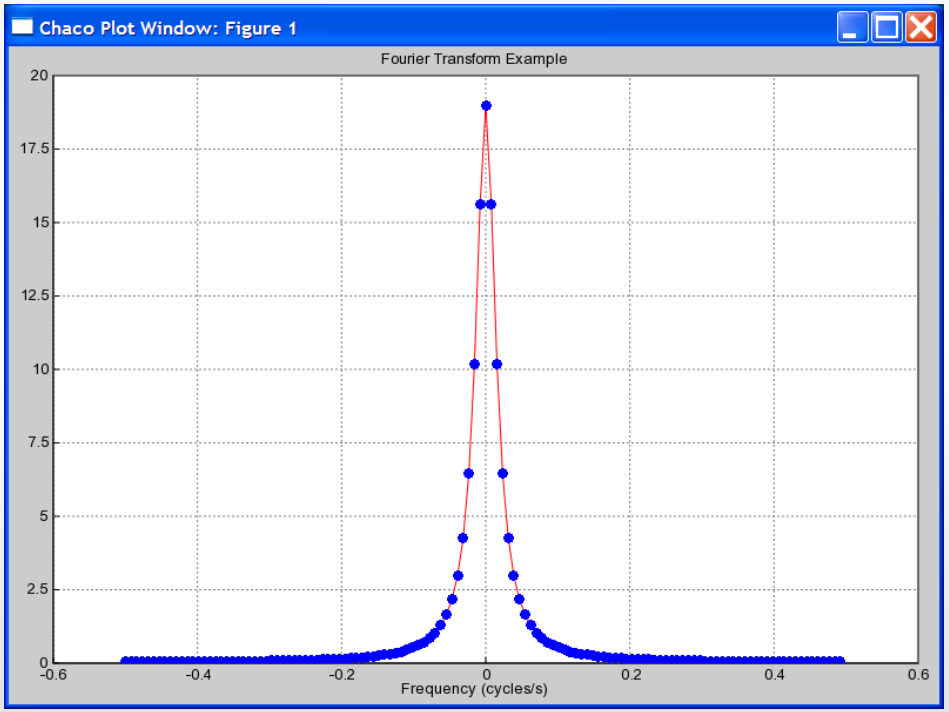
1D Spline Interpolation

```
from scipy.interpolate import interp1d
from pylab import plot, axis, legend
from numpy import linspace

# sample values
x = linspace(0,2*pi,6)
y = sin(x)

# Create a spline class for interpolation.
# kind=5 sets to 5th degree spline.
# kind=0 -> zeroth order hold.
# kind=1 or 'linear' -> linear interpolation
# kind=2 or
spline_fit = interp1d(x,y,kind=5)
xx = linspace(0,2*pi, 50)
yy = spline_fit(xx)

# display the results.
plot(xx, sin(xx), 'r-', x,y,'ro',xx,yy, 'b--',linewidth=2)
axis('tight')
legend(['actual sin', 'original samples', 'interpolated curve'])
```



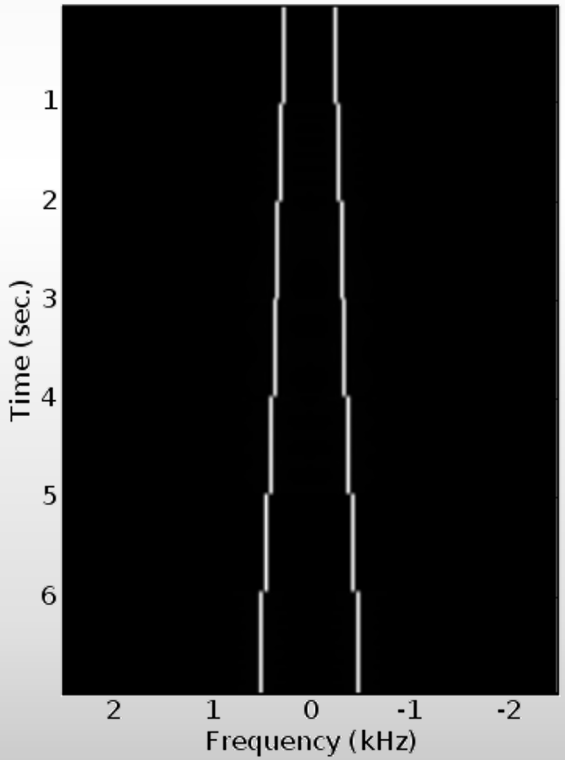
scipy.fft --- FFT and related functions

```
>>> n = fftfreq(128)*128
>>> f = fftfreq(128)
>>> ome = 2*pi*f
>>> x = (0.9)**abs(n)
>>> X = fft(x)
>>> z = exp(1j*ome)
>>> Xexact = (0.9**2 - 1)/0.9*z /   \
...          (z-0.9) / (z-1/0.9)
>>> f = fftshift(f)
>>> plot(f, fftshift(X.real),'r-',
...      f, fftshift(Xexact.real),'bo')
>>> title('Fourier Transform Example')
>>> xlabel('Frequency (cycles/s)')
>>> axis(-0.6,0.6, 0, 20)
```



EXAMPLE --- Short-Time Windowed Fourier Transform

```
rate, data = read('scale.wav')
dT, T_window = 1.0/rate, 50e-3
N_window = int(T_window * rate)
N_data = len(data)
window = get_window('hamming', N_window)
result, start = [], 0
# compute short-time FFT for each block
while (start < N_data – N_window):
        end = start + N_window
    val = fftshift(fft(window*data[start:end]))
    result.append(val)
    start = end
lastval = fft(window*data[-N_window:])
result.append(fftshift(lastval))
result = array(result,result[0].dtype)
```



Signal Processing

scipy.signal --- Signal and Image Processing

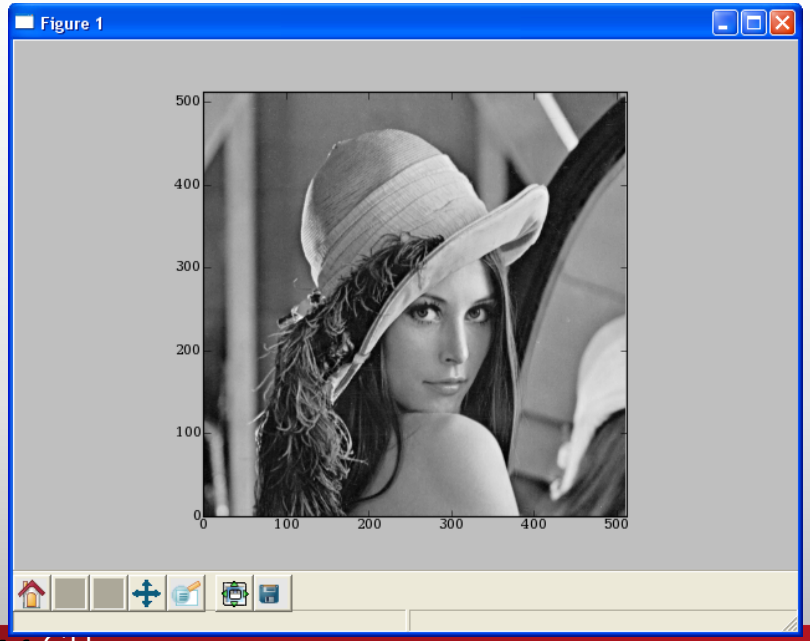
What's Available?

- Filtering
  - General 2-D Convolution (more boundary conditions)
  - N-D convolution
  - B-spline filtering
  - N-D Order filter, N-D median filter, faster 2d version,
  - IIR and FIR filtering and filter design
- LTI systems
  - System simulation
  - Impulse and step responses
  - Partial fraction expansion

Image Processing

```
# The famous lena image is packaged with scipy
>>> from scipy import misc.lena, signal
>>> lena = lena().astype(float32)
>>> imshow(lena, cmap=cm.gray)
# Blurring using a median filter
>>> fl = signal.medfilt2d(lena, [15,15])
>>> imshow(fl, cmap=cm.gray)
```

LENA IMAGE



MEDIAN FILTERED IMAGE

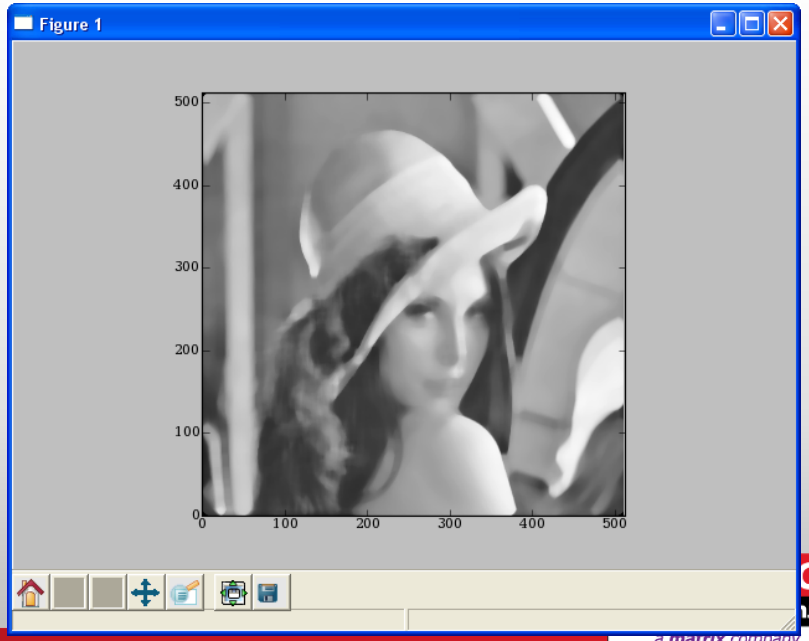
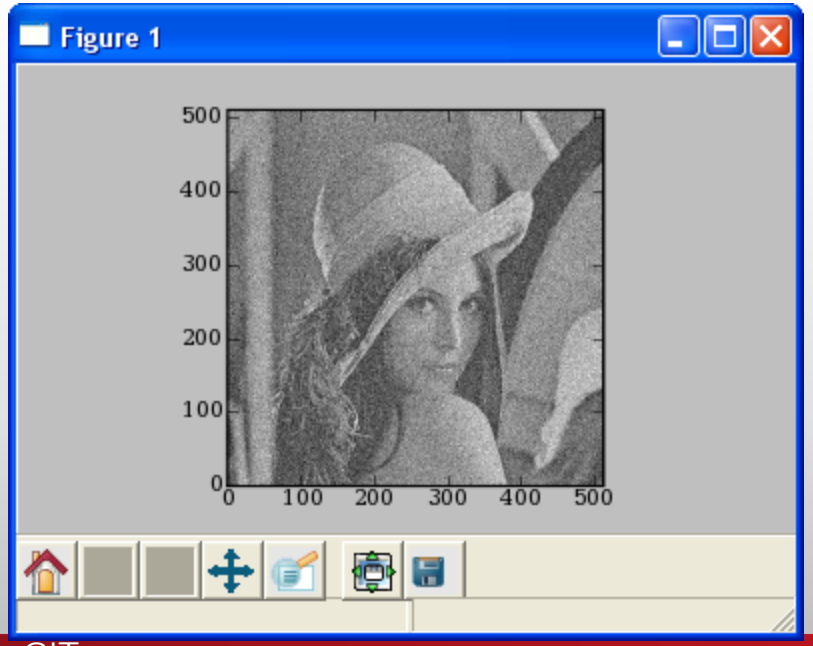


Image Processing

```
# Noise removal using wiener filter
>>> from scipy.stats import norm
>>> ln = lena + norm(0,32).rvs(lena.shape)
>>> imshow(ln)
>>> cleaned = signal.wiener(ln)
>>> imshow(cleaned)
```

NOISY IMAGE



FILTERED IMAGE

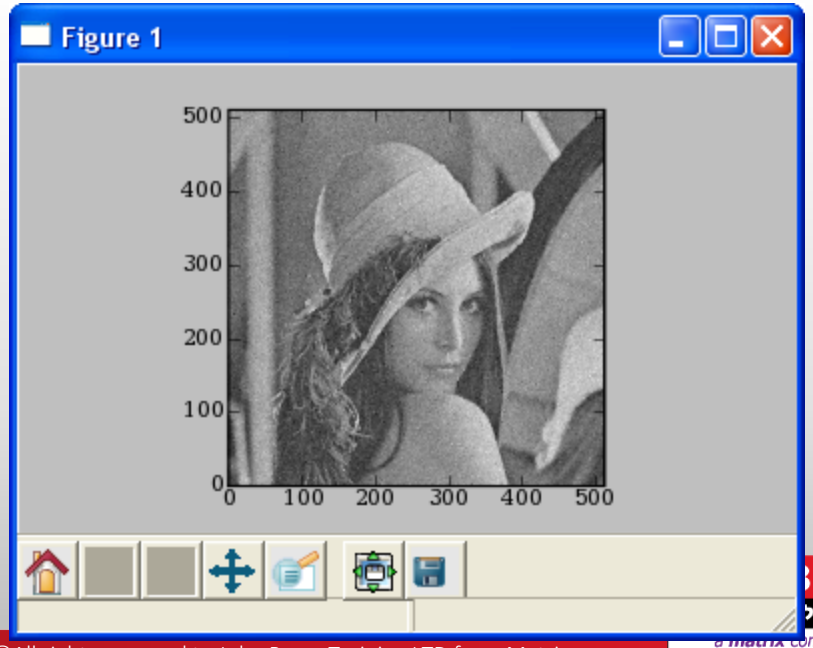
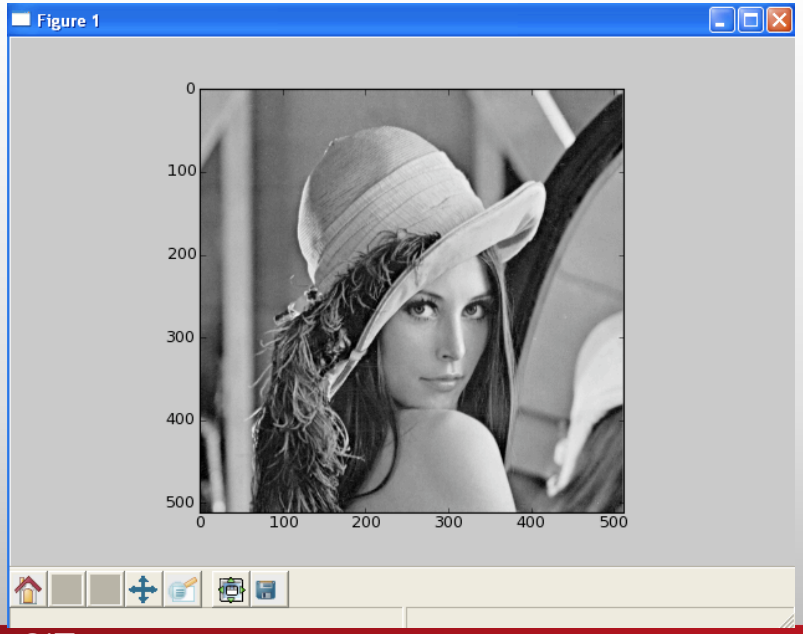


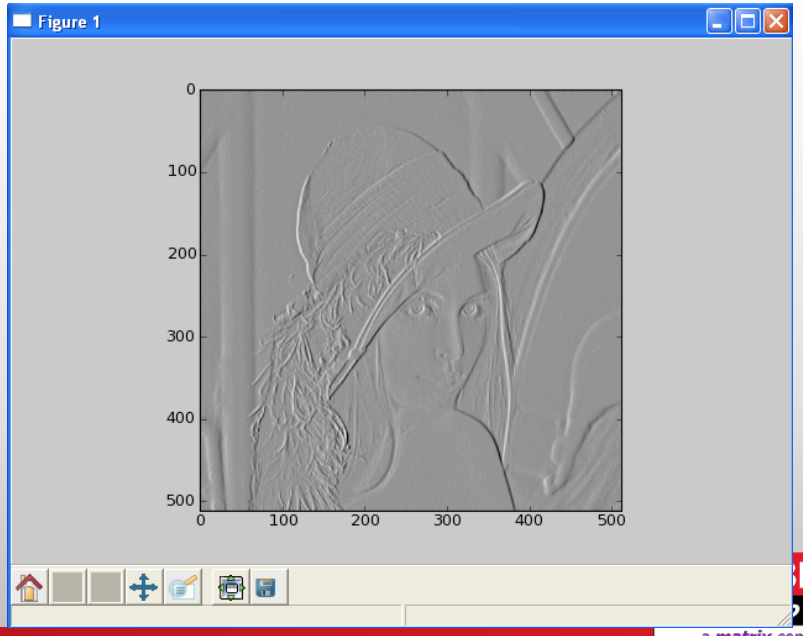
Image Processing

```
# Edge detection using Sobel filter
>>> from scipy.ndimage.filters import sobel
>>> imshow(lena)
>>> edges = sobel(lena)
>>> imshow(edges)
```

NOISY IMAGE



FILTERED IMAGE



LTI Systems

```
>>> b,a = [1],[1,6,25]
>>> ltisys = signal.lti(b,a)
>>> t,h = ltisys.impulse()
>>> ts,s = ltisys.step()
>>> plot(t,h,ts,s)
>>> legend(['Impulse response','Step response'])
```

$$H(s) = \frac{1}{s^2 + 6s + 25}$$

Optimization

scipy.optimize --- unconstrained minimization and root finding

- Unconstrained Optimization

  `fmin` (Nelder-Mead simplex), `fmin_powell` (Powell's method), `fmin_bfgs` (BFGS quasi-Newton method), `fmin_ncg` (Newton conjugate gradient), `leastsq` (Levenberg-Marquardt), `anneal` (simulated annealing global minimizer), `brute` (brute force global minimizer), brent (excellent 1-D minimizer), golden, bracket

- Constrained Optimization

  `fmin_l_bfgs_b`, `fmin_tnc` (truncated newton code), `fmin_cobyla` (constrained optimization by linear approximation), `fminbound` (interval constrained 1-d minimizer)

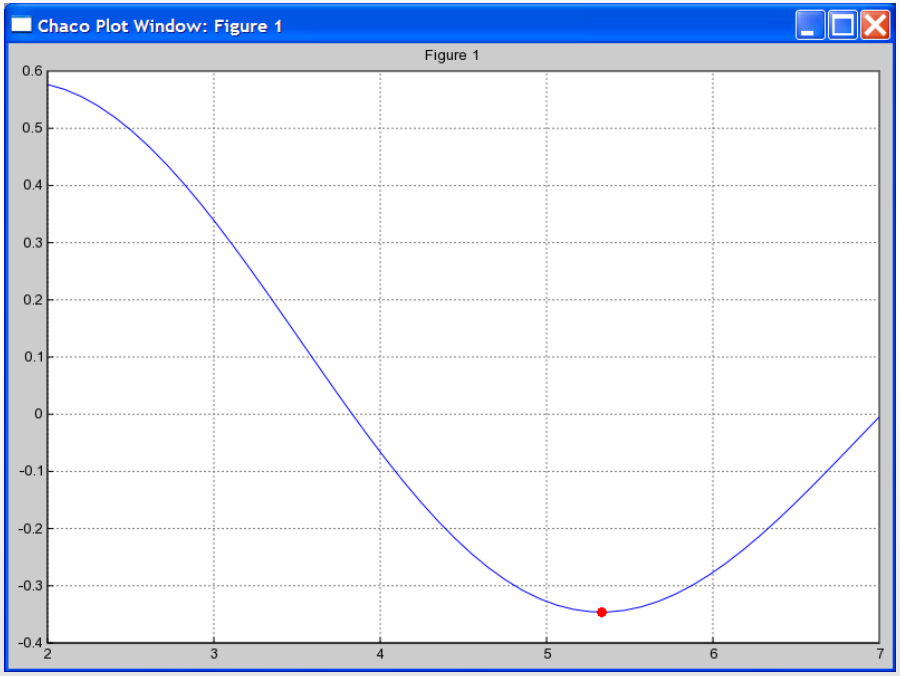
- Root finding

  `fsolve` (using MINPACK), `brentq`, `brenth`, `ridder`, `newton`, `bisect`, `fixed_point` (fixed point equation solver)

Optimization

EXAMPLE: MINIMIZE BESSEL FUNCTION

```
# minimize 1st order bessel
# function between 4 and 7
>>> from scipy.special import j1
>>> from scipy.optimize import \
      fminbound

>>> x = r_[2:7.1:.1]
>>> j1x = j1(x)
>>> plot(x,j1x,'-')
>>> hold(True)
>>> x_min = fminbound(j1,4,7)
>>> j1_min = j1(x_min)
>>> plot([x_min],[j1_min],'ro')
```



Optimization

EXAMPLE: SOLVING NONLINEAR EQUATIONS

Solve the non-linear equations

$$
\begin{aligned}
3x_0 - \cos(x_1 x_2) + a &= 0 \\
x_0^2 - 81(x_1 + 0.1)^2 + \sin(x_2) + b &= 0 \\
e^{-x_0 x_1} + 20x_2 + c &= 0
\end{aligned}
$$

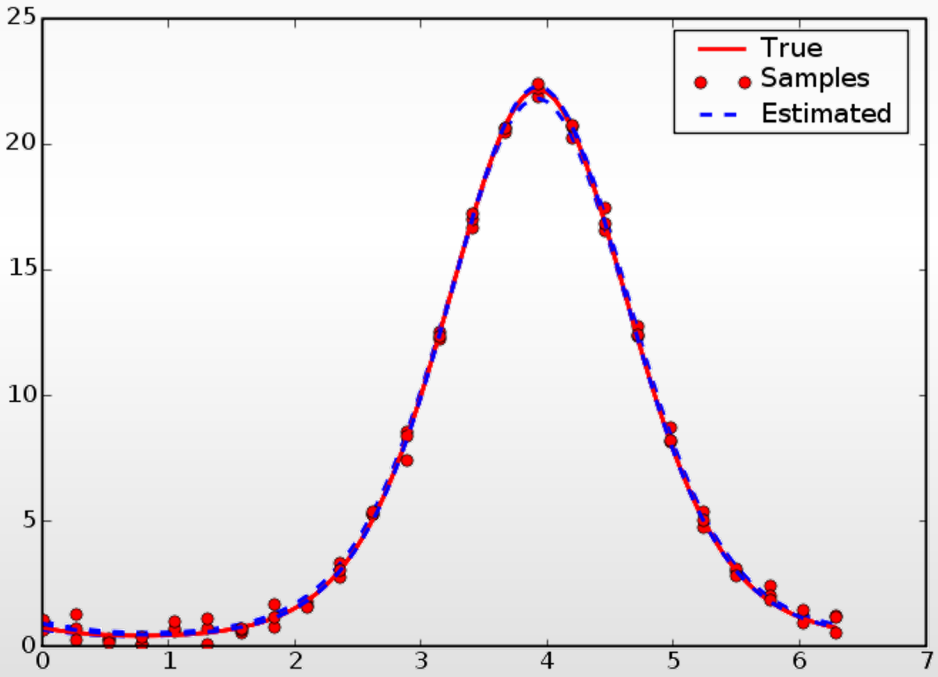
```
>>> def nonlin(x,a,b,c):
>>>     x0,x1,x2 = x
>>>     return [3*x0-cos(x1*x2)+ a,
>>>             x0*x0-81*(x1+0.1)**2
                + sin(x2)+b,
>>>             exp(-x0*x1)+20*x2+c]
>>> a,b,c = -0.5,1.06,(10*pi-3.0)/3
>>> root = optimize.fsolve(nonlin,   [0.1,0.1,-
0.1],args=(a,b,c))
>>> print root
[ 0.5     0.     -0.5236]
>>> print nonlin(root,a,b,c)
[0.0, -2.231104190e-12, 7.46069872e-14]
```

starting location for search

Optimization

EXAMPLE: Non-linear least-squares data fitting

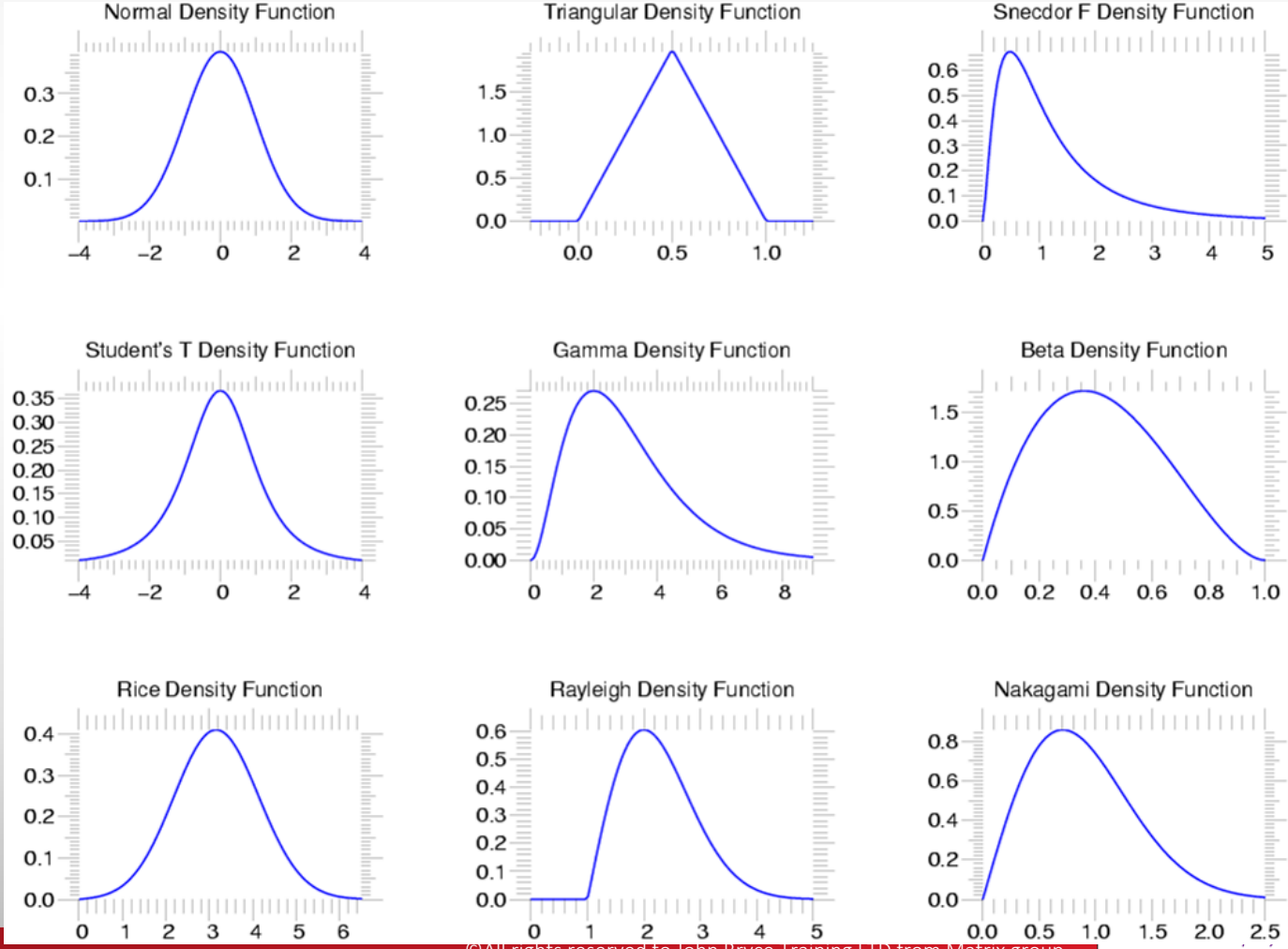
```
# fit data-points to a curve
# demo/data_fitting/datafit.py
>>> from numpy.random import randn
>>> from numpy import exp, sin, pi
>>> from numpy import linspace
>>> from scipy.optimize import leastsq
>>> def func(x,A,a,f,phi):
      return A*exp(-a*sin(f*x+pi/4))
>>> def errfunc(params, x, data):
      return func(x, *params) - data
>>> ptrue = [3,2,1,pi/4]
>>> x = linspace(0,2*pi,25)
>>> true = func(x, *ptrue)
>>> noisy = true + 0.3*randn(len(x))
>>> p0 = [1,1,1,1]
>>> pmin, ier = leastsq(errfunc, p0,
                        args=(x, noisy))
>>> pmin
array([3.1705, 1.9501, 1.0206, 0.7034])
```



Statistics

scipy.stats --- CONTINUOUS DISTRIBUTIONS

over 80 continuous distributions!



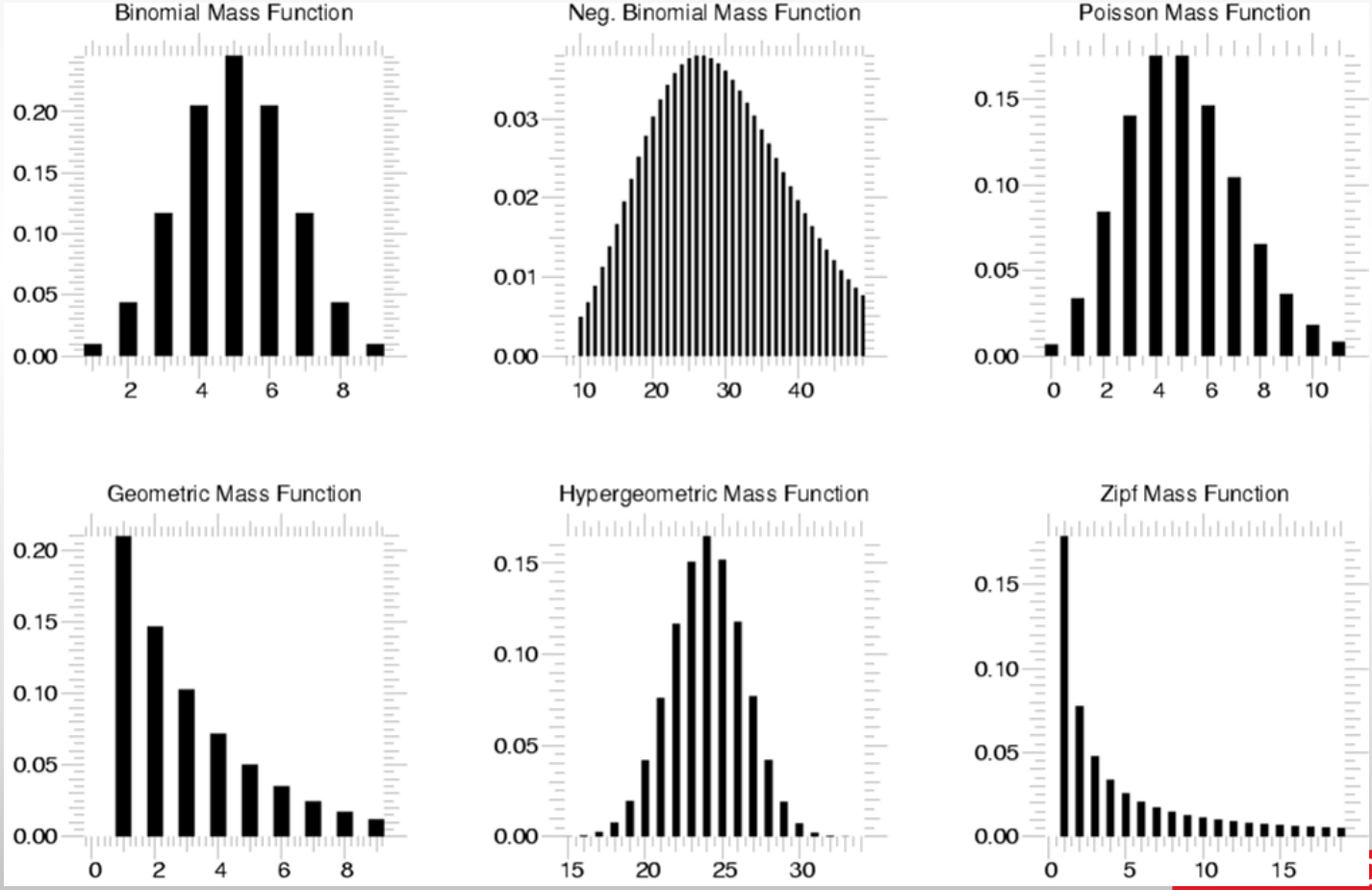
METHODS

- pdf
- cdf
- rvs
- ppf
- stats

Statistics

scipy.stats --- Discrete Distributions

10 standard discrete distributions (plus any arbitrary finite RV)



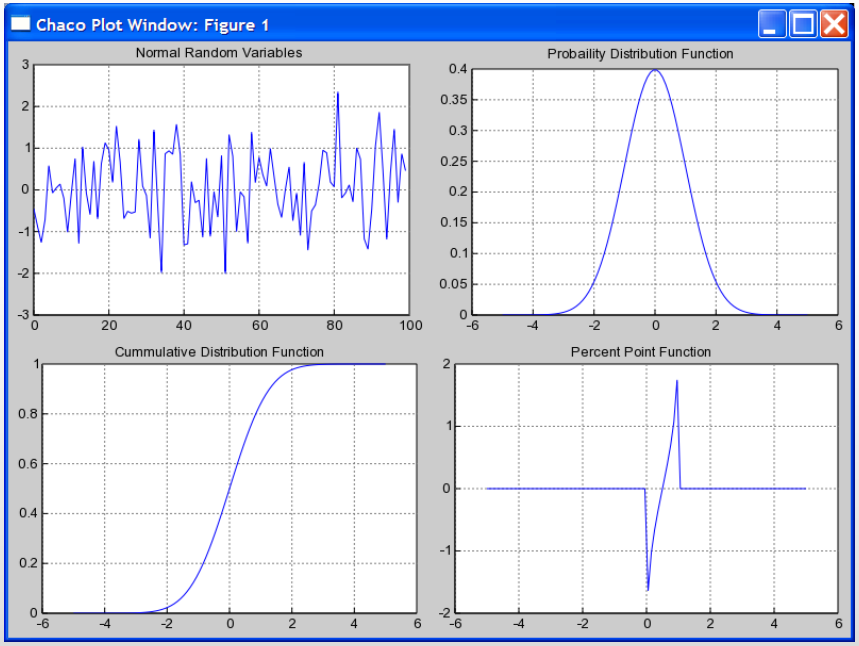
METHODS

- pdf
- cdf
- rvs
- ppf
- stats

Using stats objects

DISTRIBUTIONS

```
# Sample normal dist. 100 times.
>>> samp = stats.norm.rvs(size=100)

>>> x = r_[-5:5:100j]
# Calculate probability dist.
>>> pdf = stats.norm.pdf(x)
# Calculate cummulative Dist.
>>> cdf = stats.norm.cdf(x)
# Calculate Percent Point Function
>>> ppf = stats.norm.ppf(x)
```



Statistics

scipy.stats --- Basic Statistical Calculations on Data

- numpy.mean, numpy.std, numpy.var, numpy.cov
- stats.skew, stats.kurtosis, stats.moment

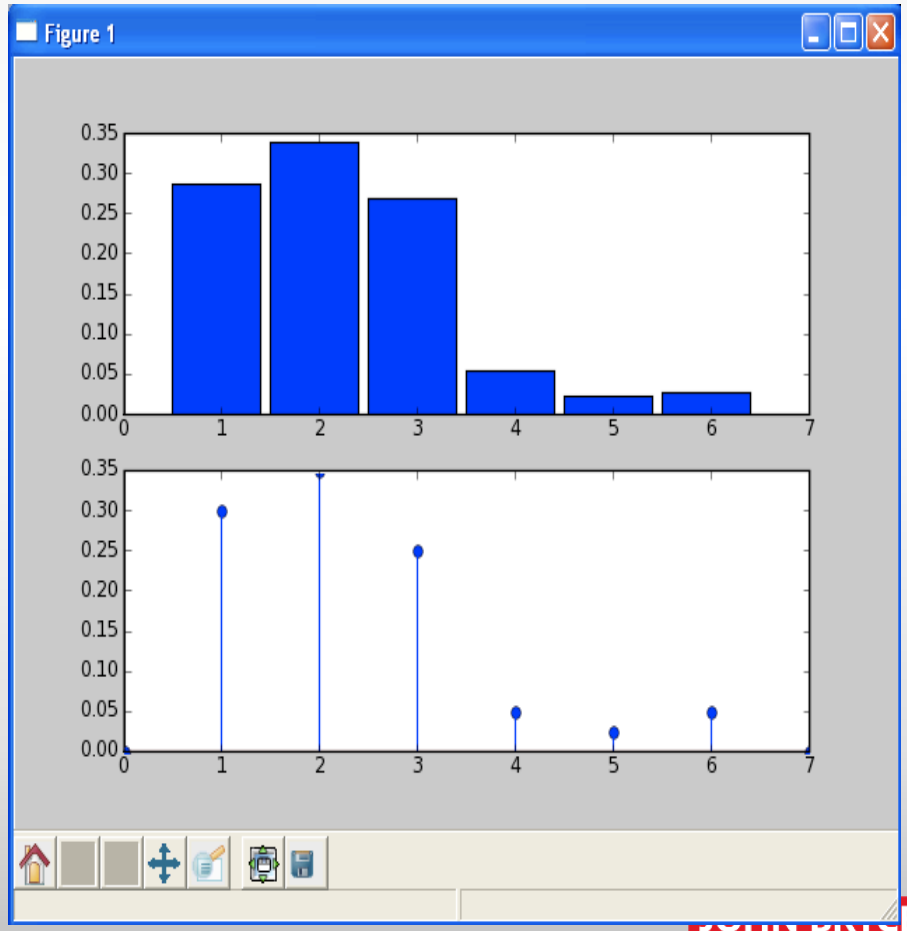
scipy.stats.bayes_mvs --- Bayesian mean, variance, and std.

```
# Create “frozen” Gamma distribution with a=2.5
>>> grv = stats.gamma(2.5)
>>> grv.stats()     # Theoretical mean and variance
(array(2.5), array(2.5))
# Estimate mean, variance, and std with 95% confidence
>>> vals = grv.rvs(size=100)
>>> stats.bayes_mvs(vals, alpha=0.95)
((2.52887906081, (2.19560839724, 2.86214972438)),
 (2.87924964268, (2.17476164549, 3.8070215789)),
 (1.69246760584, (1.47470730841, 1.95115903475)))
# (expected value and confidence interval for each of
# mean, variance, and standard-deviation)
```

Using stats objects

CREATING NEW DISCRETE DISTRIBUTIONS

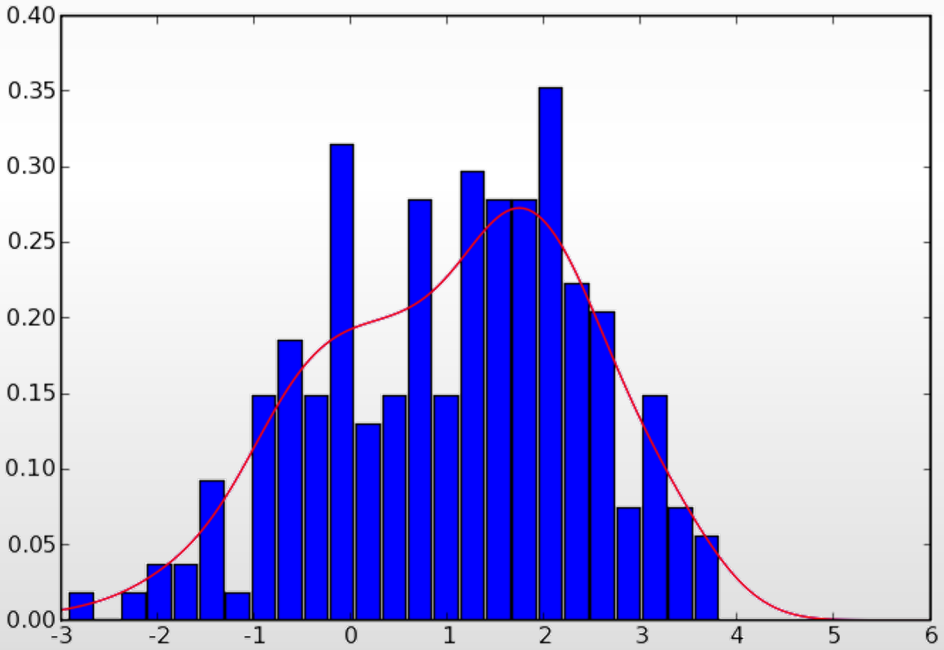
```
# Create a loaded dice.
>>> from scipy.stats import rv_discrete
>>> xk = [1,2,3,4,5,6]
>>> pk = [0.3,0.35,0.25,0.05,
                0.025,0.025]
>>> new = rv_discrete(name='loaded',
                                values=(xk,pk))

# Calculate histogram
>>> samples = new.rvs(size=1000)
>>> bins=linspace(0.5,5.5,6)
>>> subplot(211)
>>> hist(samples,bins=bins,normed=True)
# Calculate pmf
>>> x = range(0,8)
>>> subplot(212)
>>> stem(x,new.pmf(x))
```



Statistics

Continuous PDF Estimation using Gaussian Kernel Density Estimation

```
# Sample normal dist. 100 times.
>>> rv1 = stats.norm()
>>> rv2 = stats.norm(2.0,0.8)
>>> samp = r_[rv1.rvs(size=100),
              rv2.rvs(size=100)]
# Kernel estimate (smoothed histogram)
>>> apdf = stats.kde.gaussian_kde(samp)
>>> x = linspace(-3,6,200)
>>> plot(x, apdf(x),'r')

# Histogram
>>> hist(x, bins=25, normed=True)
```



Linear Algebra

scipy.linalg --- FAST LINEAR ALGEBRA

- Uses ATLAS if available --- very fast
- Low-level access to BLAS and LAPACK routines in modules linalg.fblas, and linalg.flapack (FORTRAN order)
- High level matrix routines
  - Linear Algebra Basics: inv, solve, det, norm, lstsq, pinv
  - Decompositions: eig, lu, svd, orth, cholesky, qr, schur
  - Matrix Functions: expm, logm, sqrtm, cosm, coshm, funm (general matrix functions)

Linear Algebra

LU FACTORIZATION

```
>>> from scipy import linalg
>>> a = array([[1,3,5],
...            [2,5,1],
...            [2,3,6]])
# time consuming factorization
>>> lu, piv = linalg.lu_factor(a)

# fast solve for 1 or more
# right hand sides.
>>> b = array([10,8,3])
>>> linalg.lu_solve((lu, piv), b)
array([-7.82608696,  4.56521739,
        0.82608696])
```

EIGEN VALUES AND VECTORS

```
>>> from scipy import linalg
>>> a = array([[1,3,5],
...            [2,5,1],
...            [2,3,6]])
# compute eigen values/vectors
>>> vals, vecs = linalg.eig(a)
# print eigen values
>>> vals
array([ 9.39895873+0.j,
       -0.73379338+0.j,
        3.33483465+0.j])
# eigen vectors are in columns
# print first eigen vector
>>> vecs[:,0]
array([-0.57028326,
       -0.41979215,
       -0.70608183])
# norm of vector should be 1.0
>>> linalg.norm(vecs[:,0])
1.0
```

Matrix Objects

STRING CONSTRUCTION

```
>>> from numpy import mat
>>> a = mat('[1,3,5;2,5,1;2,3,6]')
>>> a
matrix([[1, 3, 5],
        [2, 5, 1],
        [2, 3, 6]])
```

TRANSPOSE ATTRIBUTE

```
>>> a.T
matrix([[1, 2, 2],
        [3, 5, 3],
        [5, 1, 6]])
```

INVERTED ATTRIBUTE

```
>>> a.I
matrix([[-1.1739, 0.1304,  0.956],
        [ 0.4347, 0.1739, -0.391],
        [ 0.1739, -0.130,  0.0434]
       ])
```

DIAGONAL

```
>>> a.diagonal()
matrix([[1, 5, 6]])
>>> a.diagonal(-1)
matrix([[3, 1]])
```

SOLVE

```
>>> b = mat('10;8;3')
>>> a.I*b
matrix([[-7.82608696],
        [ 4.56521739],
        [ 0.82608696]])

>>> from scipy import linalg
>>> linalg.solve(a,b)
matrix([[-7.82608696],
        [ 4.56521739],
        [ 0.82608696]])
```

Integration

scipy.integrate --- General purpose Integration

- Ordinary Differential Equations (ODE)

  integrate.odeint, integrate.ode

- Samples of a 1-d function

  integrate.trapz (trapezoidal Method), integrate.simps (Simpson Method), integrate.romb (Romberg Method)

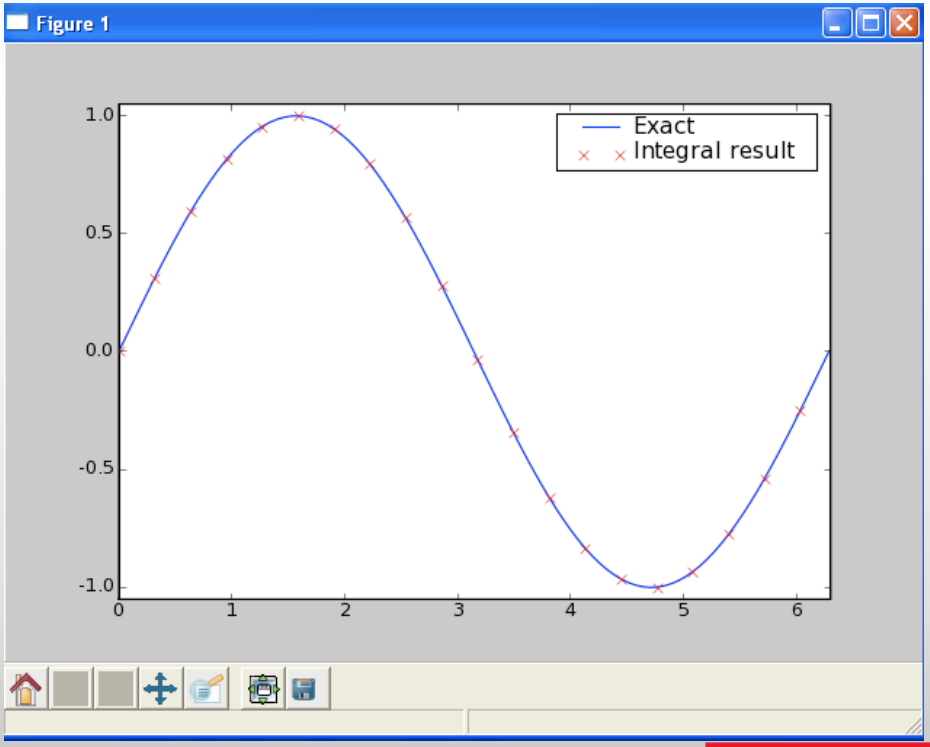
- Arbitrary callable function

  integrate.quad (general purpose), integrate.dblquad (double integration), integrate.tplquad (triple integration), integrate.fixed_quad (fixed order Gaussian integration), integrate.quadrature (Gaussian quadrature to tolerance), integrate.romberg (Romberg)

Integration

scipy.integrate --- Example

```
# Compare sin to integral(cos)
>>> def func(x):
    return integrate.quad(cos,0,x)[0]
>>> vecfunc = vectorize(func)

>>> x = r_[0:2*pi:100j]
>>> x2 = x[::5]
>>> y = sin(x)
>>> y2 = vecfunc(x2)
>>> plot(x,y,x2,y2,'rx')
>>> legend(['Exact',
...         'Integral Result'])
```



Special Functions

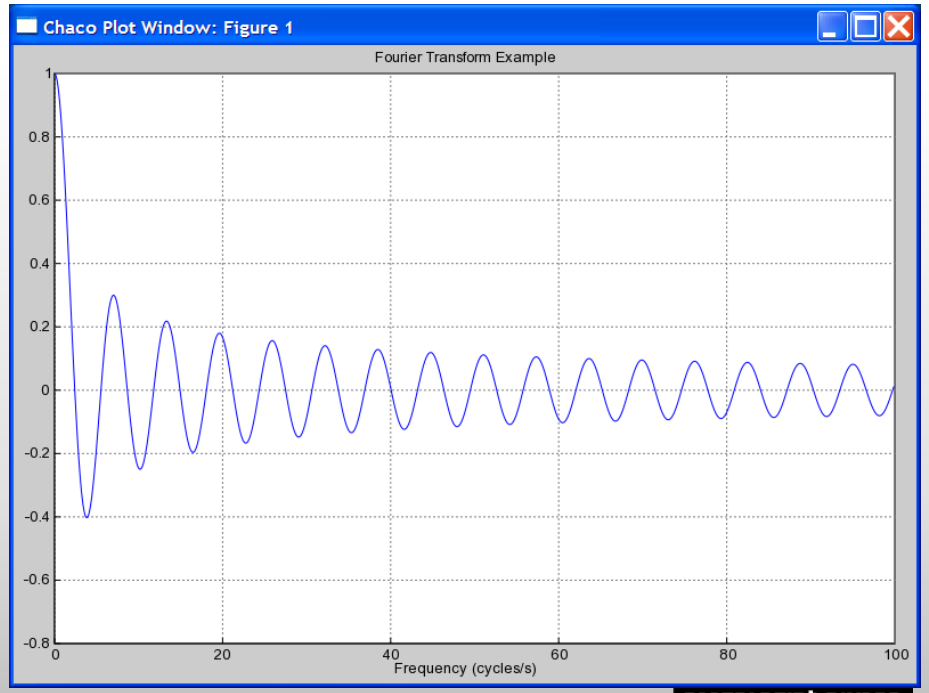
scipy.special

Includes over 200 functions:

Airy, Elliptic, Bessel, Gamma, HyperGeometric, Struve, Error, Orthogonal Polynomials, Parabolic Cylinder, Mathieu, Spheroidal Wave, Kelvin

FIRST ORDER BESSEL EXAMPLE

```
>>> from scipy import special
>>> x = r_[0:100:0.1]
>>> j0x = special.j0(x)
>>> plot(x,j0x)
```



Pandas

Overview

- Click to edit Master text styles
  - Second level
    - Third level
      - Fourth level
        - Fifth level

Pandas is well suited for:

Tabular Data (SQL table & Excel spreadsheet)

Ordered and Unordered Time Series Data

Arbitrary Matrix Data

Any other form of observational / statistical data sets

Series

- Click to edit Master text styles
  - Second level
    - Third level
      - Fourth level
        - Fifth level

Series

```
In [2]: import pandas as pd

In [3]: ls=['avi', 'dani', 'rina']

In [4]: s1 = pd.Series(ls)

In [5]: s1
Out[5]:
0     avi
1    dani
2    rina
dtype: object
```

Adding labels

```
In [10]: ls2=['avi', 'dani', 'rina']

In [11]: ls1 = [1000, 2000, 3000]

In [12]: s2 = pd.Series(ls1,ls2)

In [13]: s2
Out[13]:
avi     1000
dani    2000
rina    3000
dtype: int64
```

Using Dictionary

```
In [14]: d=dict(avi=2000, dani=3000, rina=4000)

In [15]: d
Out[15]: {'avi': 2000, 'dani': 3000, 'rina': 4000}

In [16]: pd.Series(d)
Out[16]:
avi     2000
dani    3000
rina    4000
dtype: int64
```

Operations

```
In [36]: d1
Out[36]: {'avi': 2000, 'dani': 3000, 'rina': 4000}

In [37]: d2
Out[37]: {'avi': 4000, 'dani': 2000, 'dina': 2000, 'rina': 5000}

In [38]: s1=pd.Series(d1)

In [39]: s2=pd.Series(d2)
```

```
In [43]: s1['avi']
Out[43]: 2000

In [44]: s1
Out[44]:
avi     2000
dani    3000
rina    4000
dtype: int64

In [45]: s2
Out[45]:
avi     4000
dani    2000
dina    2000
rina    5000
dtype: int64

In [46]: s1+s2
Out[46]:
avi     6000.0
dani    5000.0
dina       NaN
rina    9000.0
dtype: float64
```

Handling Nulls

```
In [71]: s3 = s1+s2

In [72]: s3
Out[72]:
avi     6000.0
dani    5000.0
dina       NaN
rina    9000.0
dtype: float64

In [73]: s3.isnull()
Out[73]:
avi     False
dani    False
dina     True
rina    False
dtype: bool

In [74]: s3.notnull()
Out[74]:
avi      True
dani     True
dina    False
rina     True
dtype: bool
```

```
In [75]: s4=s3[s3.notnull()]

In [76]: s4
Out[76]:
avi     6000.0
dani    5000.0
rina    9000.0
dtype: float64
```

Dataframe

- Click to edit Master text styles
  - Second level
    - Third level
      - Fourth level
        - Fifth level

Operations

- Click to edit Master text styles
  - Second level
    - Third level
      - Fourth level
        - Fifth level

DataFrame From NumPy Array

```
import pandas as pd
import numpy as np
```

```
samp=np.random.randint(100, 600,size=(4,5))
```

```
samp
```

```
array([[205, 225, 129, 549, 328],
       [150, 348, 325, 474, 268],
       [495, 348, 488, 579, 371],
       [407, 158, 478, 120, 575]])
```

```
df=pd.DataFrame(samp,index=['avi','dani','rina','dina'],
                          columns=['Jan','Feb','Mar','Apr','May'])
```

	Jan	Feb	Mar	Apr	May
avi	205	225	129	549	328
dani	150	348	325	474	268
rina	495	348	488	579	371
dina	407	158	478	120	575

Selecting and Indexing

```
df['Jan']
```

```
avi      205
dani     150
rina     495
dina     407
Name: Jan, dtype: int64
```

```
df['Jan']['avi']
```

```
205
```

```
# Pass a list of column names
df[['Jan','May']]
```

	Jan	May
avi	205	328
dani	150	268
rina	495	371
dina	407	575

Using SQL Syntax

```
df.Feb
```

```
avi     558
dani    336
rina    168
dina    571
Name: Feb, dtype: int64
```

```
df.Feb.avi
```

```
558
```

Not Recommended

```
df.Feb[df.Feb>500]
```

```
avi     558
dina    571
Name: Feb, dtype: int64
```

Types

Its important to understand the types and the permitted operations on that types

```
type(df['May'])
```

```
pandas.core.series.Series
```

```
type(df)
```

```
pandas.core.frame.DataFrame
```

```
type(df['May']>300)
```

```
pandas.core.series.Series
```

```
type(df['May']['avi'])
```

```
numpy.int64
```

Unique Data

```
df['Jan']['rina'] = 300
df['Jan']['avi'] = 300
```

```
df
```

	Jan	Feb	Mar	Apr	May
avi	300	455	367	207	440
dani	276	169	213	561	510
rina	300	423	533	411	314
dina	399	257	592	356	215

```
df['Jan'].unique()
```

```
array([300, 276, 399])
```

```
df['Jan'].nunique()
```

```
3
```

```
df['Jan'].value_counts()
```

```
300    2
276    1
399    1
Name: Jan, dtype: int64
```

Update Data

```
def add_bonus(x):
    return x*1.2
df.apply(add_bonus)
```

	Jan	Feb	Mar	Apr	May
avi	360.0	546.0	440.4	248.4	528.0
dani	331.2	202.8	255.6	673.2	612.0
rina	360.0	507.6	639.6	493.2	376.8
dina	478.8	308.4	710.4	427.2	258.0

```
df['Jan'].apply(lambda x:x+1000)
```

```
avi     1300
dani    1276
rina    1300
dina    1399
Name: Jan, dtype: int64
```

```
df['Jan'].sum()
```

```
1275L
```

Adding New Data

Column

```
df['Expected'] = df['May'] + 20
```

```
df
```

	Jan	Feb	Mar	Apr	May	Expected
avi	205	225	129	549	328	348
dani	150	348	325	474	268	288
rina	495	348	488	579	371	391
dina	407	158	478	120	575	595

Row

```
df.loc['eli'] = [100,200,300,400,500,600]
```

```
df
```

	Jan	Feb	Mar	Apr	May	Expected
avi	500	558	140	589	560	580
dani	177	336	405	267	277	297
rina	506	168	269	161	217	237
dina	237	571	138	598	273	293
eli	100	200	300	400	500	600

Removing Data

```
df.drop('rina',axis=0)
```

	Jan	Feb	Mar	Apr	May	Expected
avi	205	225	129	549	328	348
dani	150	348	325	474	268	288
dina	407	158	478	120	575	595

```
# Not inplace unless specified!
df
```

	Jan	Feb	Mar	Apr	May	Expected
avi	205	225	129	549	328	348
dani	150	348	325	474	268	288
rina	495	348	488	579	371	391
dina	407	158	478	120	575	595

```
df.drop('Apr',axis=1,inplace=True)
```

```
df
```

	Jan	Feb	Mar	May	Expected
avi	205	225	129	328	348
dani	150	348	325	268	288
rina	495	348	488	371	391
dina	407	158	478	575	595

Selecting Data

Using label

```
df.loc['avi']
```

```
Jan          205
Feb          225
Mar          129
May          328
Expected     348
Name: avi, dtype: int64
```

Using position

```
df.iloc[2]
```

```
Jan          495
Feb          348
Mar          488
May          371
Expected     391
Name: rina, dtype: int64
```

Selecting subset

```
df.loc['avi','Jan']
```

```
205
```

```
df.loc[['avi','rina'],['Feb','Apr']]
```

	Feb	Apr
avi	225.0	NaN
rina	348.0	NaN

More Selections

```
df
```

	Jan	Feb	Mar	Apr	May
avi	360	133	185	320	411
dani	132	211	228	226	422
rina	247	336	107	470	515
dina	210	491	156	282	574

```
df['Jan'][1:3]
```

```
dani    132
rina    247
Name: Jan, dtype: int64
```

```
df.loc['dani':,'Feb':'Apr']
```

	Feb	Mar	Apr
dani	211	228	226
rina	336	107	470
dina	491	156	282

```
df
```

	Jan	Feb	Mar	Apr	May
avi	360	133	185	320	411
dani	132	211	228	226	422
rina	247	336	107	470	515
dina	210	491	156	282	574

```
df.iloc[:3]
```

	Jan	Feb	Mar	Apr	May
avi	360	133	185	320	411
dani	132	211	228	226	422
rina	247	336	107	470	515

```
df.iloc[[1,3],[2,3]]
```

	Mar	Apr
dani	228	226
dina	156	282

Conditional Selection

```
df>200
```

	Jan	Feb	Mar	May	Expected
avi	True	True	False	True	True
dani	False	True	True	True	True
rina	True	True	True	True	True
dina	True	False	True	True	True

Boolean operation – returns new DF

```
df[df>200]
```

	Jan	Feb	Mar	May	Expected
avi	205.0	225.0	NaN	328	348
dani	NaN	348.0	325.0	268	288
rina	495.0	348.0	488.0	371	391
dina	407.0	NaN	478.0	575	595

Selecting based on another DF

```
df[df['Jan']>200]
```

	Jan	Feb	Mar	May	Expected
avi	205	225	129	328	348
rina	495	348	488	371	391
dina	407	158	478	575	595

Selecting based on a series

More Conditions

```
df[df['Jan']>200]['May']
```

```
avi      328
rina     371
dina     575
Name: May, dtype: int64
```

```
df[df['Jan']>200][['Jan','Mar']]
```

	Jan	Mar
avi	205	129
rina	495	488
dina	407	478

For two conditions you can use | and & with parenthesis:

```
df[(df['Jan']>200) & (df['Feb'] > 300)]
```

	Jan	Feb	Mar	May	Expected
rina	495	348	488	371	391

Using Callback

```
df.loc[lambda df:df['Jan']>200,:]
```

	Jan	Feb	Mar	Apr	May
avi	360	133	185	320	411
rina	247	336	107	470	515
dina	210	491	156	282	574

```
df.loc[:,lambda df: ['Jan','Mar']]
```

	Jan	Mar
avi	360	185
dani	132	228
rina	247	107
dina	210	156

Sorting

```
df
```

	Jan	Feb	Mar	Apr	May
avi	300	455	367	207	440
dani	276	169	213	561	510
rina	300	423	533	411	314
dina	399	257	592	356	215

```
df.sort_values('Jan')
```

	Jan	Feb	Mar	Apr	May
dani	276	169	213	561	510
avi	300	455	367	207	440
rina	300	423	533	411	314
dina	399	257	592	356	215

Use inplace=True to apply sorting

NaN values

```
df['Jan']['avi'] = np.nan
df['Feb']['dani'] = np.nan
df
```

	Jan	Feb	Mar	Apr	May
avi	NaN	455.0	367	207	440
dani	276.0	NaN	213	561	510
rina	300.0	423.0	533	411	314
dina	399.0	257.0	592	356	215

```
df.isnull()
```

	Jan	Feb	Mar	Apr	May
avi	True	False	False	False	False
dani	False	True	False	False	False
rina	False	False	False	False	False
dina	False	False	False	False	False

```
df.dropna()
```

	Jan	Feb	Mar	Apr	May
rina	300.0	423.0	533	411	314
dina	399.0	257.0	592	356	215

```
df.fillna(1000)
```

	Jan	Feb	Mar	Apr	May
avi	1000.0	455.0	367	207	440
dani	276.0	1000.0	213	561	510
rina	300.0	423.0	533	411	314
dina	399.0	257.0	592	356	215

Multi-index

```
# Index Levels
years = ['2016','2016','2016','2016','2017','2017','2017','2017']
q = [1,2,3,4,1,2,3,4]
t = list(zip(years,q))
mi = pd.MultiIndex.from_tuples(t)
```

```
mi
```

```
MultiIndex(levels=[[u'2016', u'2017'], [1, 2, 3, 4]],
           labels=[[0, 0, 0, 0, 1, 1, 1, 1], [0, 1, 2, 3, 0, 1, 2, 3]])
```

```
df = pd.DataFrame(np.random.randn(8,2),index=mi,columns=['A','B'])
df
```

		A	B
2016	**1**	-0.889180	0.311152
	2	-0.612847	-0.353895
	3	-0.866984	0.711970
	4	-0.057056	-0.564472
2017	**1**	-1.537100	0.851859
	2	0.725848	-0.918994
	3	1.189998	0.580911
	4	-1.893752	-0.996923

Multi-index

```
df.loc['2016']
```

	A	B
1	-0.889180	0.311152
2	-0.612847	-0.353895
3	-0.866984	0.711970
4	-0.057056	-0.564472

```
df.loc['2017'].loc[1]
```

```
A   -1.537100
B    0.851859
Name: 1, dtype: float64
```

```
df.index.names = ['Year','Q']
```

```
df
```

		A	B
Year	Q		
2016	1	-0.889180	0.311152
	2	-0.612847	-0.353895
	3	-0.866984	0.711970
	4	-0.057056	-0.564472
2017	1	-1.537100	0.851859
	2	0.725848	-0.918994
	3	1.189998	0.580911
	4	-1.893752	-0.996923

Multi-index

```
df.xs('2017')
```

Q	A	B
1	-1.462987	0.345033
2	0.577788	-0.654170
3	-1.341800	-0.188705
4	0.477607	0.096507

```
df.xs(['2016',1])
```

```
A     1.266636
B    -0.576670
Name: (2016, 1), dtype: float64
```

```
df.xs(1,level='Q')
```

Year	A	B
2016	1.266636	-0.576670
2017	-1.462987	0.345033

Concatenation

```
df2
```

	Jan	Feb	Mar	Apr	May
eli	213	337	252	273	352
yosi	140	180	513	547	439
rani	127	191	215	547	207
meir	343	376	269	408	179

```
df
```

	Jan	Feb	Mar	Apr	May
avi	213	337	252	273	352
dani	140	180	513	547	439
rina	127	191	215	547	207
dina	343	376	269	408	179

```
pd.concat([df,df2])
```

	Jan	Feb	Mar	Apr	May
avi	213	337	252	273	352
dani	140	180	513	547	439
rina	127	191	215	547	207
dina	343	376	269	408	179
eli	213	337	252	273	352
yosi	140	180	513	547	439
rani	127	191	215	547	207
meir	343	376	269	408	179

```
pd.concat([df,df2],axis=1)
```

	Jan	Feb	Mar	Apr	May	Jan	Feb	Mar	Apr	May
avi	213.0	337.0	252.0	273.0	352.0	NaN	NaN	NaN	NaN	NaN
dani	140.0	180.0	513.0	547.0	439.0	NaN	NaN	NaN	NaN	NaN
dina	343.0	376.0	269.0	408.0	179.0	NaN	NaN	NaN	NaN	NaN
eli	NaN	NaN	NaN	NaN	NaN	213.0	337.0	252.0	273.0	352.0
meir	NaN	NaN	NaN	NaN	NaN	343.0	376.0	269.0	408.0	179.0
rani	NaN	NaN	NaN	NaN	NaN	127.0	191.0	215.0	547.0	207.0
rina	127.0	191.0	215.0	547.0	207.0	NaN	NaN	NaN	NaN	NaN
yosi	NaN	NaN	NaN	NaN	NaN	140.0	180.0	513.0	547.0	439.0

Merging Data

```
left = pd.DataFrame({'id': [10, 20, 30, 40],
                     'Name': ['avi', 'dani', 'rina', 'dina'],
                     'City': ['haifa', 'aco', 'tel aviv', 'yafo']})

right = pd.DataFrame({'id': [10, 20, 30, 50],
                      'Salary': [1000, 2000, 3000, 4000],
                      'Position': ['CTO', 'CEO', 'COO', 'Marketing']})
```

```
1 left
```

	City	Name	id
0	haifa	avi	10
1	aco	dani	20
2	tel aviv	rina	30
3	yafo	dina	40

```
right
```

	Position	Salary	id
0	CTO	1000	10
1	CEO	2000	20
2	COO	3000	30
3	Marketing	4000	50

```
pd.merge(left,right,how='inner',on='id')
```

	City	Name	id	Position	Salary
0	haifa	avi	10	CTO	1000
1	aco	dani	20	CEO	2000
2	tel aviv	rina	30	COO	3000

```
pd.merge(left,right,how='left',on='id')
```

	City	Name	id	Position	Salary
0	haifa	avi	10	CTO	1000.0
1	aco	dani	20	CEO	2000.0
2	tel aviv	rina	30	COO	3000.0
3	yafo	dina	40	NaN	NaN

```
pd.merge(left,right,how='right',on='id')
```

	City	Name	id	Position	Salary
0	haifa	avi	10	CTO	1000
1	aco	dani	20	CEO	2000
2	tel aviv	rina	30	COO	3000
3	NaN	NaN	50	Marketing	4000

Joining

```
left = pd.DataFrame({'A': ['A0', 'A1', 'A2', 'A3'],
                     'B': ['B0', 'B1', 'B2', 'B3'],
                     'key': ['K0', 'K1', 'K0', 'K1']})

right = pd.DataFrame({'C': ['C0', 'C1'],
                      'D': ['D0', 'D1']},
                      index=['K0', 'K1'])

result = left.join(right, on='key')
```

Grouping

```
df3 = pd.DataFrame({'id': [10, 20, 30, 40],
                    'Name': ['avi', 'dani', 'rina', 'dina'],
                    'City': ['haifa', 'haifa', 'tel aviv', 'yafo']})
```

```
df3.groupby('City').count()
```

City	Name	id
haifa	2	2
tel aviv	1	1
yafo	1	1

Also:

min()
max()
std()
describe()
…

Working With Files

pd.read_csv(’cust.csv')

df.to_csv(cust.csv',index=False)

pd.read_excel(‘samp.xslx’)

df.to_excel(’samp.xlsx',sheet_name=’cust')

Also supported:

- HTML
- JSON
- …

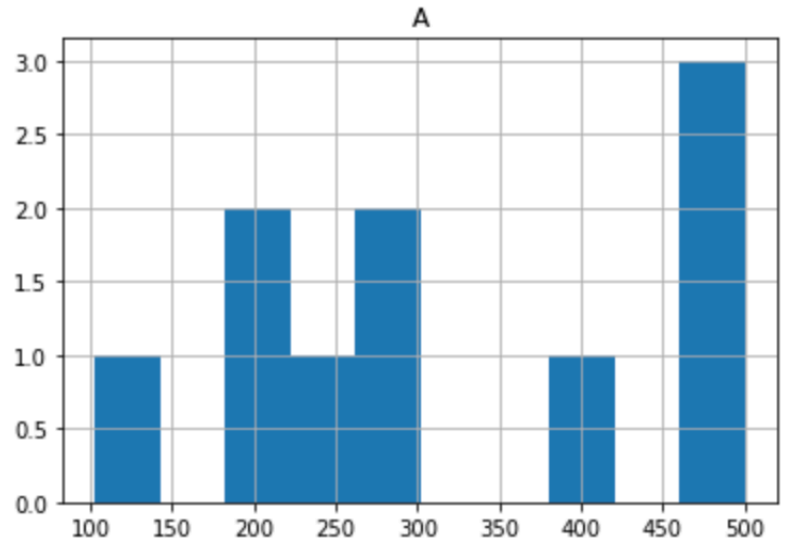
Visualization

```
%matplotlib inline
```

```
dh=pd.DataFrame(samp,columns="A,B,C,D,E,F,G,H,I,J".split(','))
```

```
dh.hist('A')
```

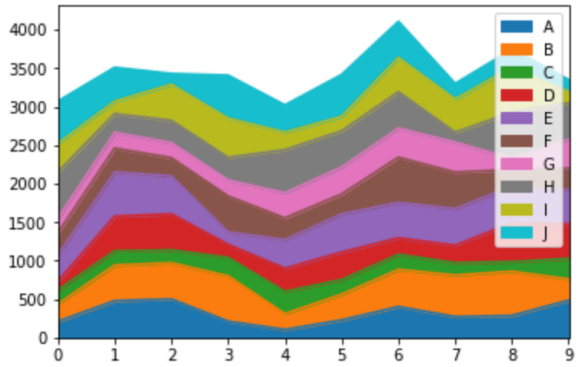
```
array([[<matplotlib.axes._subplots.AxesSubplot object at 0x119916b50>]], dtype=object)
```



More Graphs

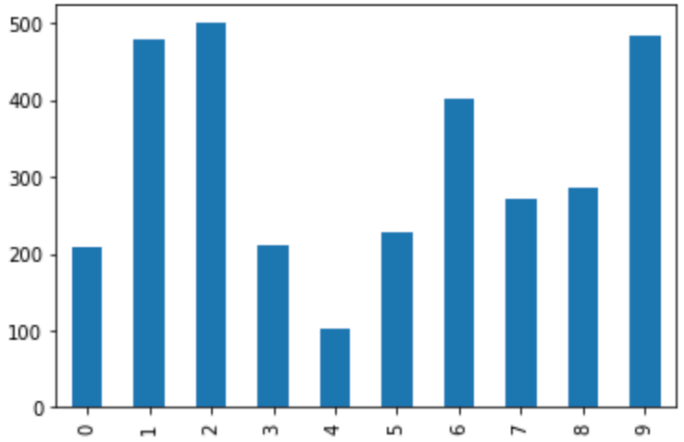
```
dh.plot.area()
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x1199d2a10>
```




```
dh['A'].plot.bar()
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x11a717950>
```



SQL Servers

The pandas.io.sql module provides a collection of query wrappers to both facilitate data retrieval and to reduce dependency on DB-specific API.

Database abstraction is provided by SQLAlchemy if installed.

In addition you will need a driver library for your database.

Examples of such drivers are

- psycopg2 for PostgreSQL
- pymysql for MySQL.
- SQLite - included in Python's standard library by default.

SQL - Function

read_sql_table(table_name, con[, schema, ...])

- Read SQL database table into a DataFrame.

read_sql_query(sql, con[, index_col, ...])

- Read SQL query into a DataFrame.

read_sql(sql, con[, index_col, ...])

- Read SQL query or database table into a DataFrame.

DataFrame.to_sql(name, con[, flavor, ...])

- Write records stored in a DataFrame to a SQL database

Example - Teradata

Install python module:

- # pip install teradata

Connect using:

- REST API (teradata.tdrest)
- ODBC (teradata.tdodbc)

```
import teradata

udaExec = teradata.UdaExec (appName="HelloWorld", version="1.0",
        logConsole=False)

session = udaExec.connect(method="odbc", system="tdprod",
        username="xxx", password="xxx");

for row in session.execute("SELECT GetQueryBand()"):
    print(row)
```

Cursors

```
import teradata
udaExec = teradata.UdaExec()
with udaExec.connect("${dataSourceName}") as session:
for row in session.execute("""SELECT InfoKey AS name, InfoData as val
            FROM DBC.DBCInfo"""):
        print(row[0] + ": " + row[1])
        print(row["name"] + ": " + row["val"])
        print(row.name + ": " + row.val)
```

```
import teradata
udaExec = teradata.UdaExec()
with udaExec.connect("${dataSourceName}") as session:
    with session.cursor() as cursor:
    for row in cursor.execute("SELECT * from ${tableName}"):
            session.execute("DELETE FROM ${tableName} WHERE id = ?", (row.id, )):
```

Stored Procedures

invoked using the “callproc” method

OUT parameters should be specified as teradata.OutParam instances

Example:

```
results = session.callproc("MyProcedure",
          (teradata.InOutParam("inputValue", "inoutVar1"),
           teradata.OutParam(),
            teradata.OutParam("outVar2", dataType="PERIOD")))

print(results.inoutVar1)

print(results.outVar1)
```

Transactions

```
import teradata
udaExec = teradata.UdaExec()
with udaExec.connect("${dataSourceName}", autoCommit=False) as session:
    session.execute("CREATE TABLE ${tableName} (${columns})")
    session.commit()
```

Seaborn

Seaborn

Statistical plotting library

Great styles

Works great with NumPy arrays and Pandas Dataframes

Seaborn

Some built in datasets

```
import seaborn as sns
%matplotlib inline
```

```
tips = sns.load_dataset('tips')
```

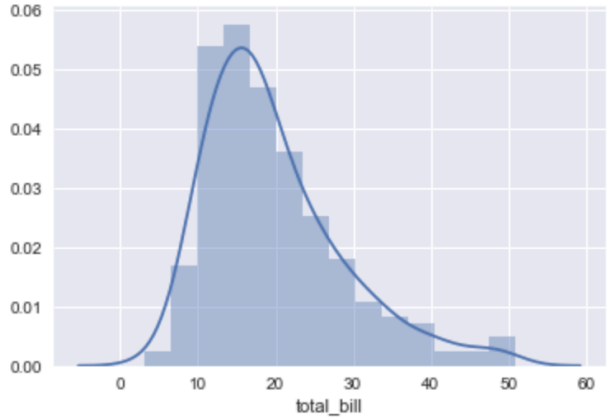
```
tips.head(10)
```

	total_bill	tip	sex	smoker	day	time	size
0	16.99	1.01	Female	No	Sun	Dinner	2
1	10.34	1.66	Male	No	Sun	Dinner	3
2	21.01	3.50	Male	No	Sun	Dinner	3
3	23.68	3.31	Male	No	Sun	Dinner	2
4	24.59	3.61	Female	No	Sun	Dinner	4
5	25.29	4.71	Male	No	Sun	Dinner	4
6	8.77	2.00	Male	No	Sun	Dinner	2
7	26.88	3.12	Male	No	Sun	Dinner	4
8	15.04	1.96	Male	No	Sun	Dinner	2
9	14.78	3.23	Male	No	Sun	Dinner	2

Distribution Plot

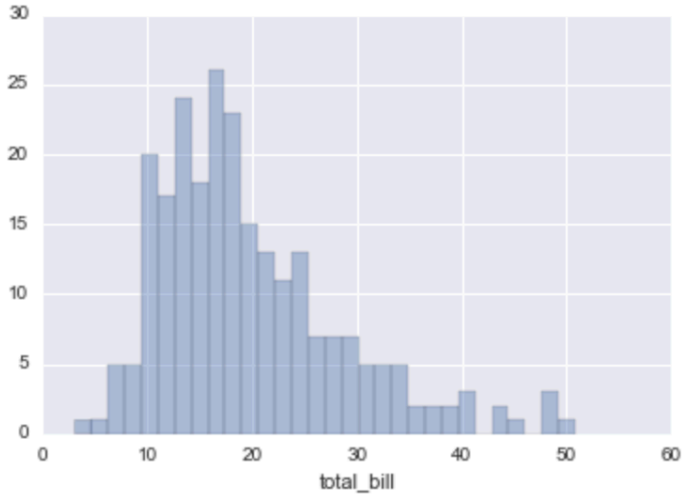
```
sns.distplot(tips['total_bill'])
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x11982a7d0>
```




```
sns.distplot(tips['total_bill'],kde=False,bins=30)
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x11c7b8668>
```



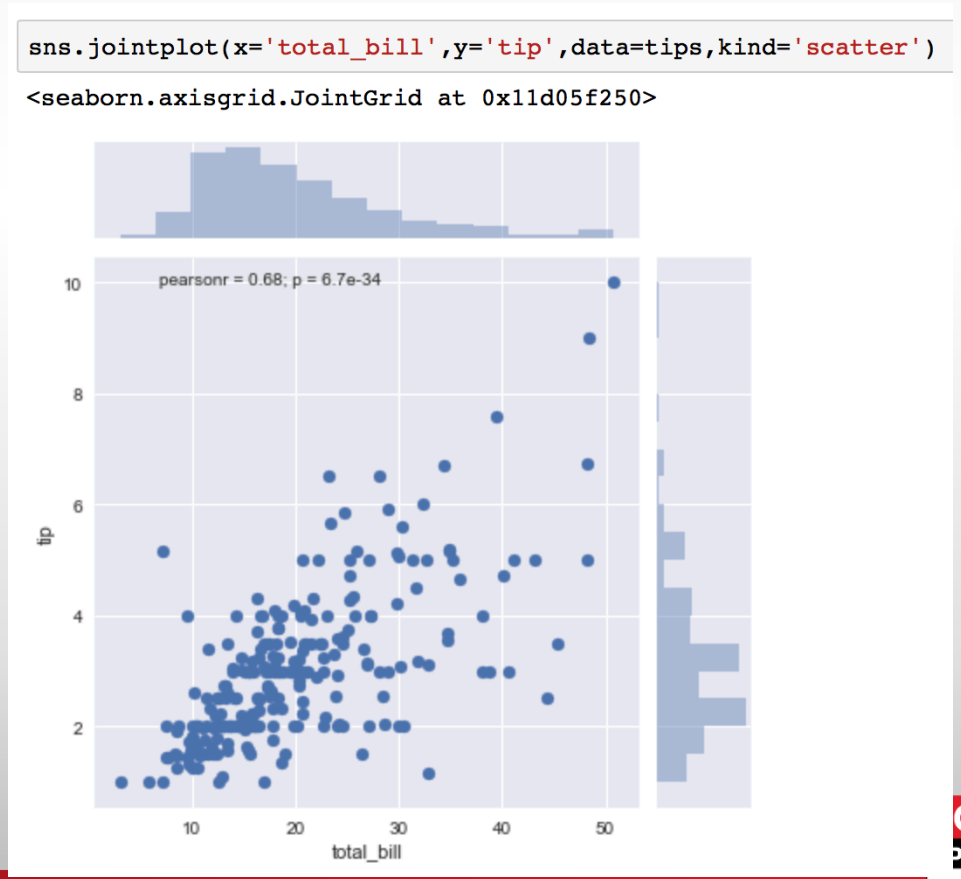
JointPlot

jointplot() allows you to basically match up two distplots() for bivariate data. With your choice of what **kind** parameter to compare with:

- scatter
- reg
- resid
- kde
- hex

```
sns.jointplot(x='total_bill',y='tip',data=tips,kind='scatter')
```

```
<seaborn.axisgrid.JointGrid at 0x11d05f250>
```

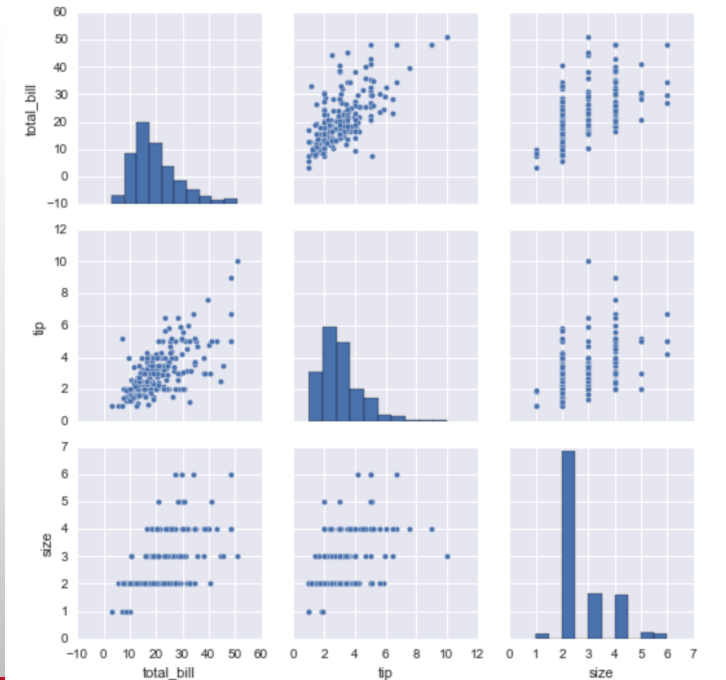


Pairplot

pairplot will plot pairwise relationships across an entire dataframe (for the numerical columns) and supports a color hue argument (for categorical columns)

```
sns.pairplot(tips)
```

```
<seaborn.axisgrid.PairGrid at 0x11e844208>
```



Categorical Data Plots

Factorplot

boxplot

violinplot

stripplot

swarmplot

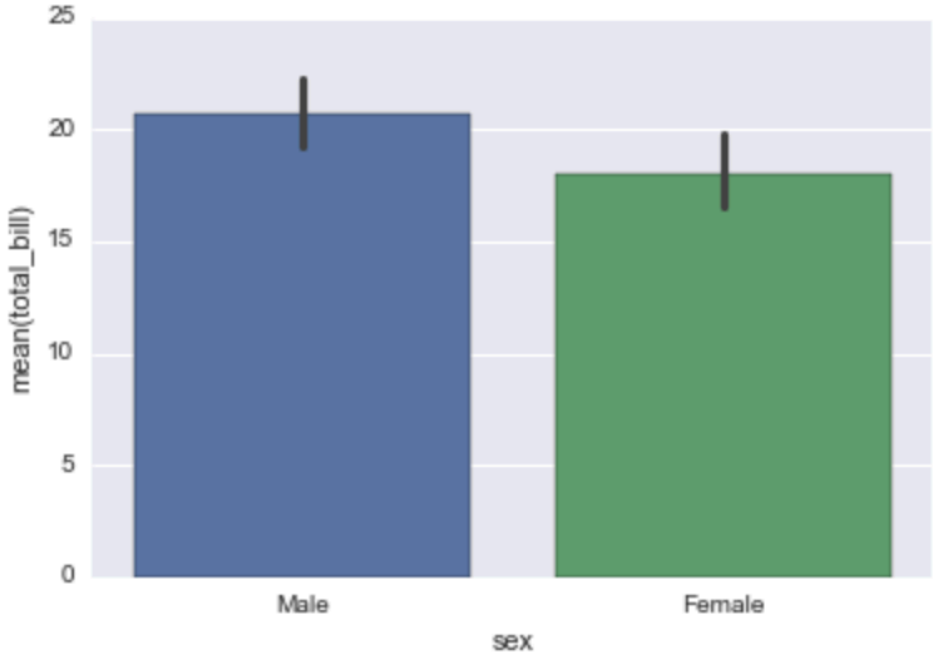
barplot

countplot

Barplot

```
sns.barplot(x='sex',y='total_bill',data=tips)
```

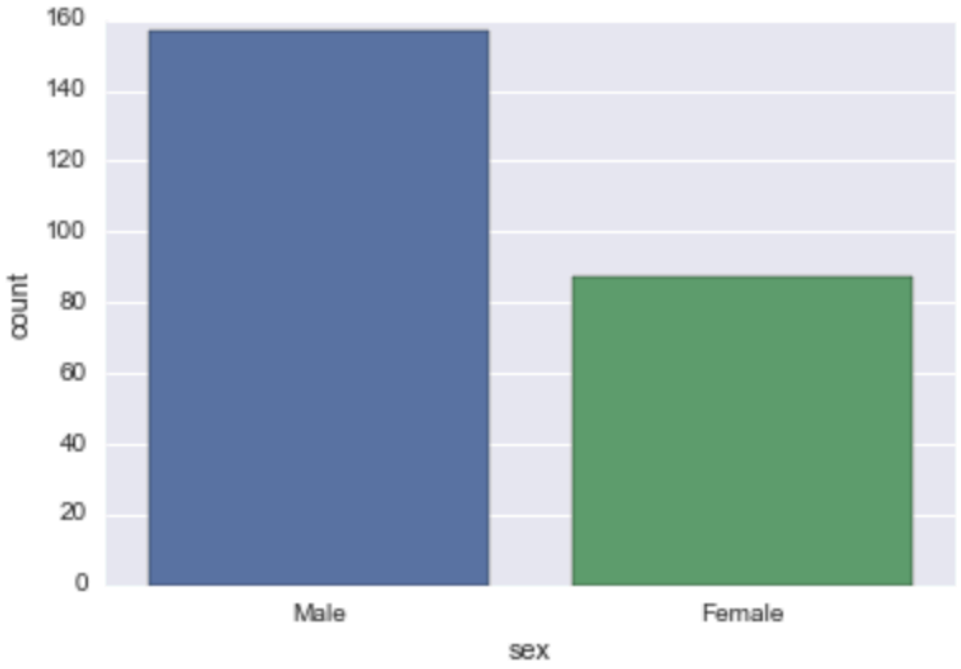
```
<matplotlib.axes._subplots.AxesSubplot at 0x11c99b8d0>
```



countplot

```
sns.countplot(x='sex',data=tips)
```

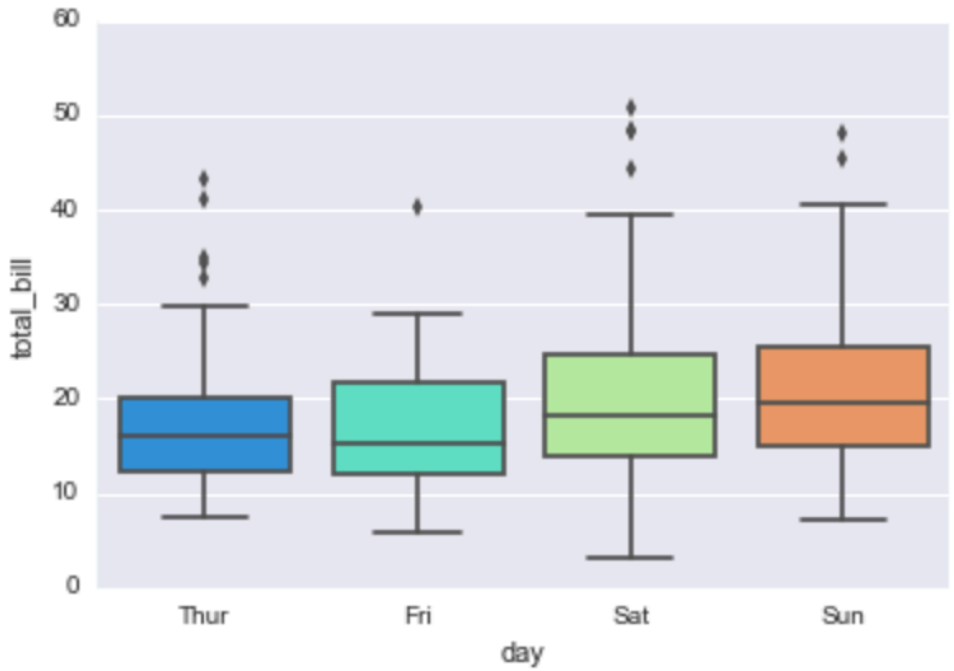
```
<matplotlib.axes._subplots.AxesSubplot at 0x1153276d8>
```



Boxplot

```
sns.boxplot(x="day", y="total_bill", data=tips,palette='rainbow')
```

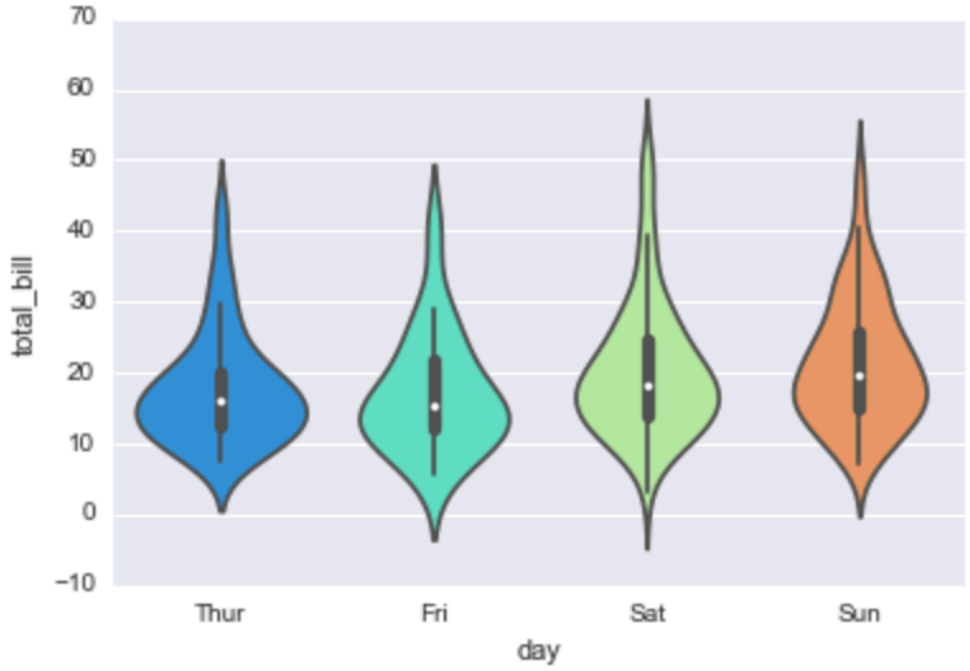
```
<matplotlib.axes._subplots.AxesSubplot at 0x11db81630>
```



voilinplot

```
sns.violinplot(x="day", y="total_bill", data=tips,palette='rainbow')
```

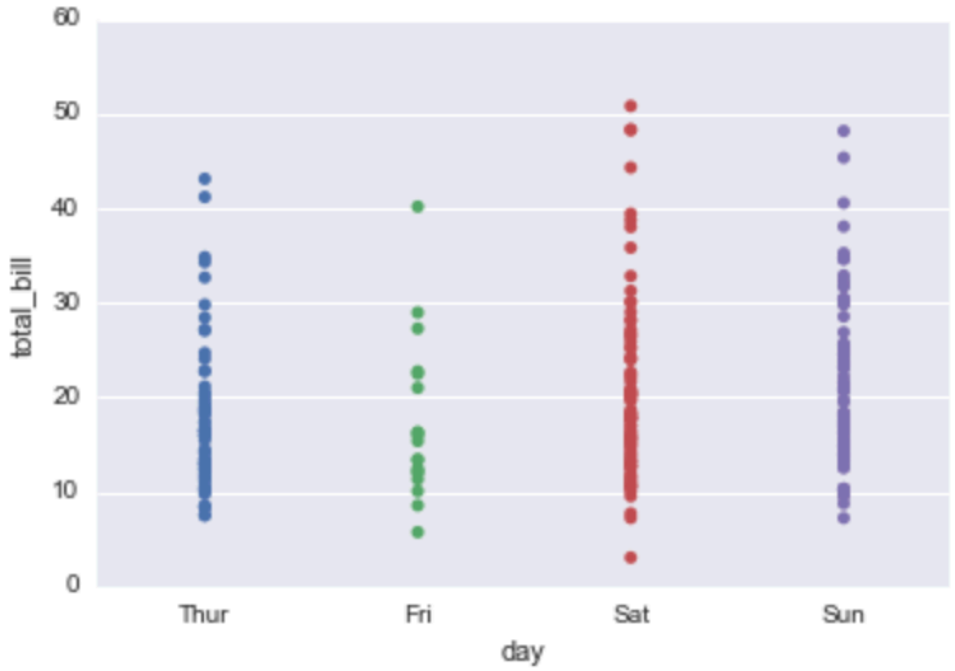
```
<matplotlib.axes._subplots.AxesSubplot at 0x11e682ba8>
```



stripplot

```
sns.stripplot(x="day", y="total_bill", data=tips)
```

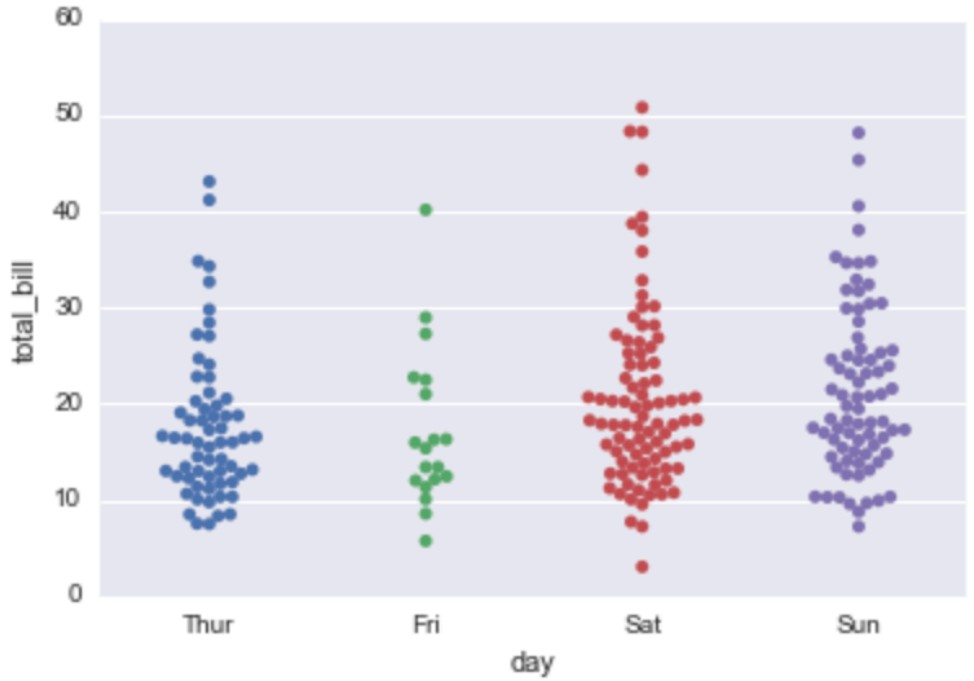
```
<matplotlib.axes._subplots.AxesSubplot at 0x120272278>
```



Swarmplot

```
sns.swarmplot(x="day", y="total_bill", data=tips)
```

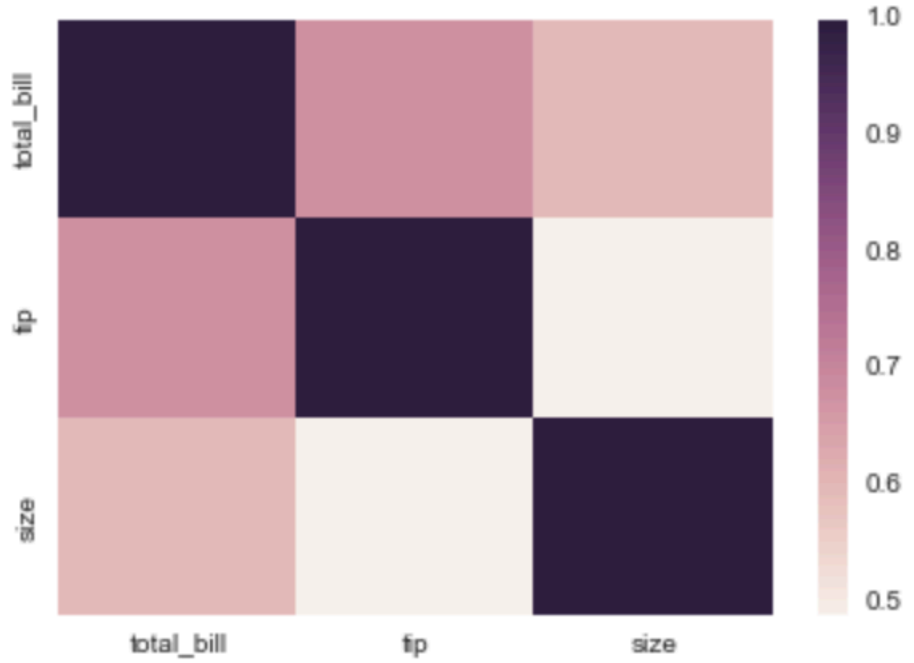
```
<matplotlib.axes._subplots.AxesSubplot at 0x120c463c8>
```



Matrix Plots

```
sns.heatmap(tips.corr())
```

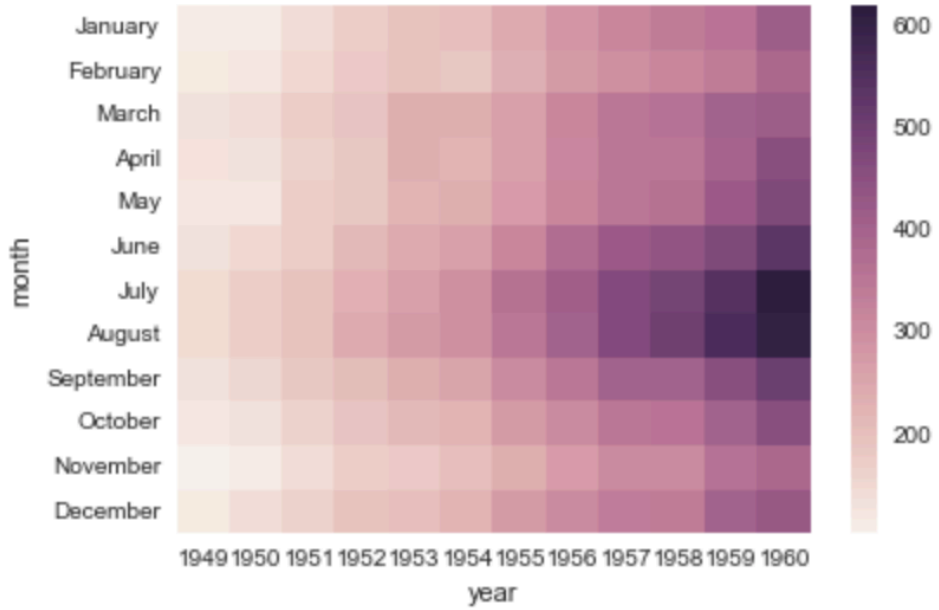
```
<matplotlib.axes._subplots.AxesSubplot at 0x11c31d470>
```



heatmap

```
pvflights = flights.pivot_table(values='passengers',index='month',columns='year')
sns.heatmap(pvflights)
```

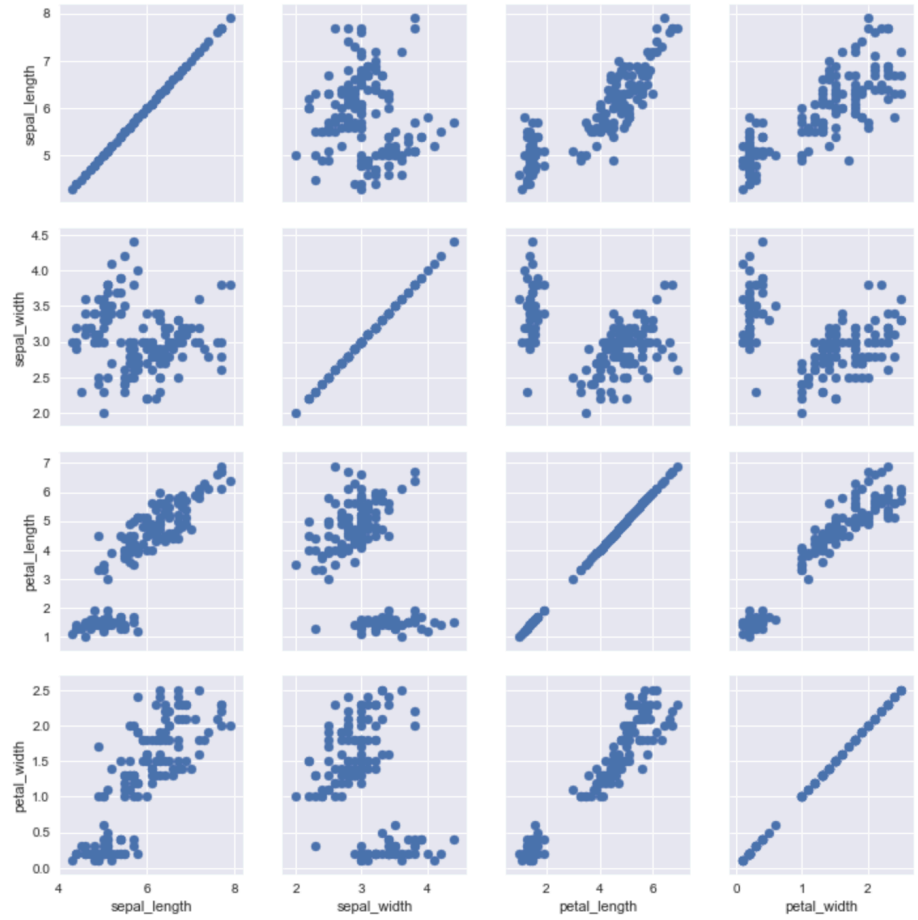
```
<matplotlib.axes._subplots.AxesSubplot at 0x11cd09320>
```



Grids

```
g = sns.PairGrid(iris)
g.map(plt.scatter)
```

```
<seaborn.axisgrid.PairGrid at 0x10e227a10>
```



Regression Plots

```
sns.lmplot(x='total_bill',y='tip',data=tips)
```

```
<seaborn.axisgrid.FacetGrid at 0x117c81048>
```