Brief Python

Python Course for Programmers

Learn Python Language for Data Science

CHAPTER 8: DATA RETRIEVAL AND DATA ANALYSIS

DR. ERIC CHOU

IEEE SENIOR MEMBER

Objectives

- Overview of Matplotlib + Numpy = PyLab
- Simple Y-Plot
- Sinusoidal Functions
- Figure Style Design
- Finding Roots
- Interactive Mode
- Cheat Sheet

A Hierarchy of Open-Source Python Libraries

- **NumPy** adds vectors, matrices and many high-level mathematical functions
- **Scipy** adds mathematical classes and functions useful to scientists.
- **MatPlotLib** adds an object-oriented API for plotting
- **PyLab** combines the other libraries to provide Matlab-like interface

Overview

LECTURE 1

Introduction

- **Matplotlib** is an excellent **2D** and **3D** graphics library for generating scientific figures. Some of the many advantages of this library include:
  - Easy to get started
  - Support for LATEX formatted labels and texts
  - Great control of every element in a figure, including figure size and **DPI**.
  - High-quality output in many formats, including **PNG**, **PDF**, **SVG**, **EPS**, and **PGF**.
  - **GUI** for interactively exploring figures and support for headless generation of figure les (useful for batch jobs).

https://matplotlib.org/tutorials/index.html

Simple Figures using **pylab**

- pylab is a module in **Matplotlib**. It can be used to perform most features of **Matlab** tool.
- We will start our tutorial on **Matplotlib** from using **pylab**.

Matplotlib
http://matplotlib.sourceforge.net/

- A python 2D plotting library
  - Publication quality figures in several hardcopy formats and interactive environments across platforms
- Can be used in
  - Python scripts,
  - the Python and IPython shell (a la matlab or mathematica),
  - web application servers, and
  - 6 GUI toolkits
- Toolkits (4)
  - http://matplotlib.sourceforge.net/users/toolkits.html
  - E.g., Basemap plots data on map projections (with continental and political boundaries)

matplotlib.pyplot vs. pylab

- Package **`matplotlib.pyplot`** provides a MATLAB-like plotting framework
- Package **`pylab`** combines **pyplot** with **NumPy** into a single namespace
  - Convenient for interactive work
  - For programming, it’s recommended that the namespaces be kept separate
- See such things as
    - **`import pylab`**
  - Or
    - **`import matplotlib.pyplot as plt`**
    - The standard alias
    - **`import numpy as np`**
    - The standard alias

matplotlib.pyplot vs. pylab

- Also
  - `from pylab import *`
- Or
  - `from matplotlib.pyplot import *`
  - `from numpy import *`

- We'll use
  - `from pylab import *`
  - Some examples don't show this—just assume it

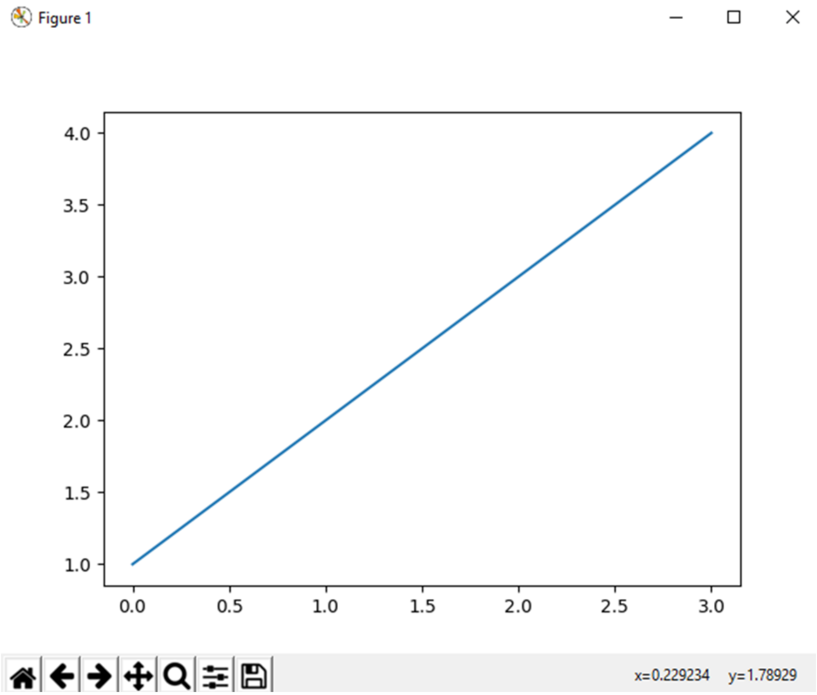
Simple Y-Plot

LECTURE 2

Simple Y-plot

Demo Program: simple_list.py

```
from pylab import *
plot([1,2,3,4])   # plot 4 y-value vs i
show()
```



`plot()`

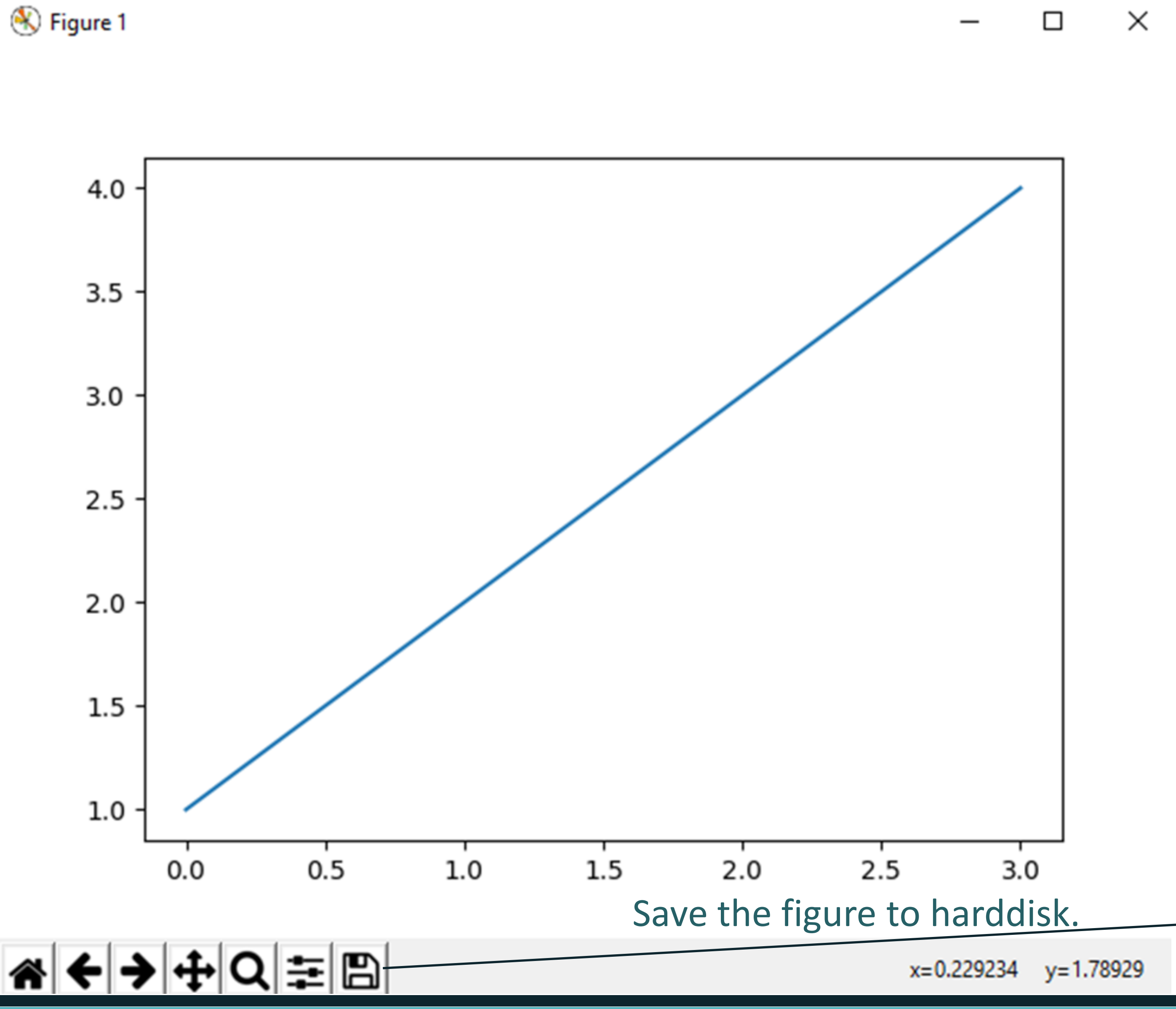
- Given a single list or array, **`plot()`** assumes it's a vector of y-values
- Automatically generates an x vector of the same length with consecutive integers beginning with 0
  - Here **`[0,1,2,3]`**
- To override default behavior, supply the x data: **`plot(`*`x`*`,`*`y`*` )`**
  - where ***x*** and ***y*** have equal lengths

`show()`

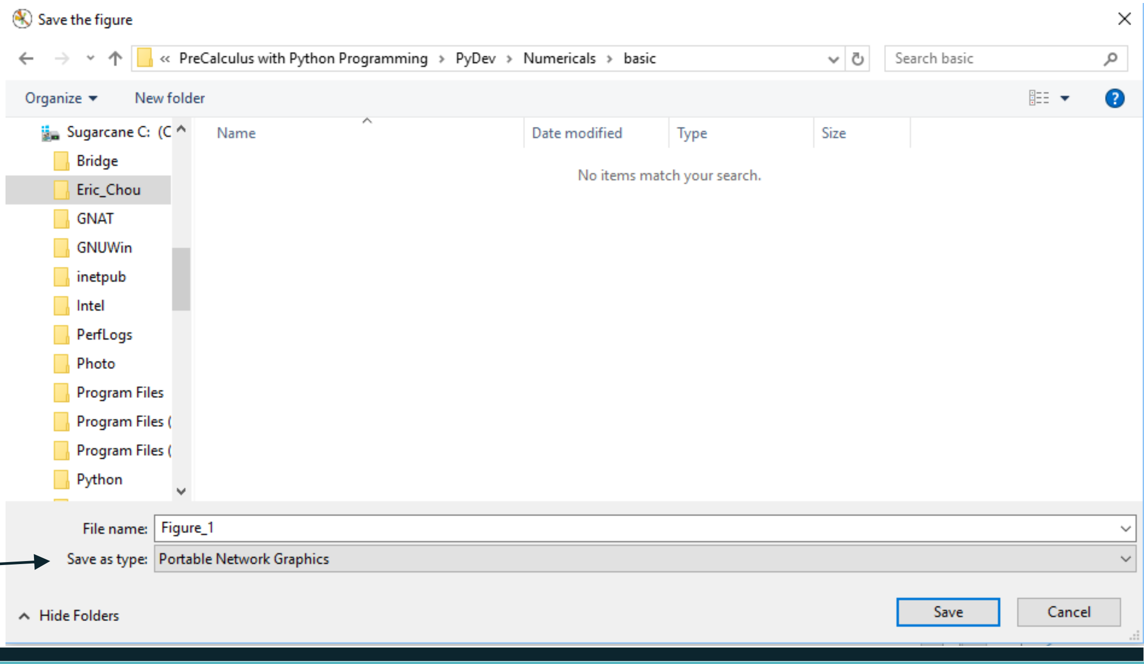
- Should be called at most once per script
  - Last line of the script
  - Then the GUI takes control, rendering the figure

show()

- In place of **`show()`**, can save the figure to with, say,
    - **`savefig('fig1.png')`**
  - Saved in the same folder as the script
- Override this with a full pathname as argument—e.g.,
    - **`savefig('E:\\SomeOtherFolder\\fig1.png')`**
- Supported formats: emf, eps, pdf, png, ps, raw, rgba, svg, svgz
  - If no extension specified, defaults to .png



Save the figure to harddisk.



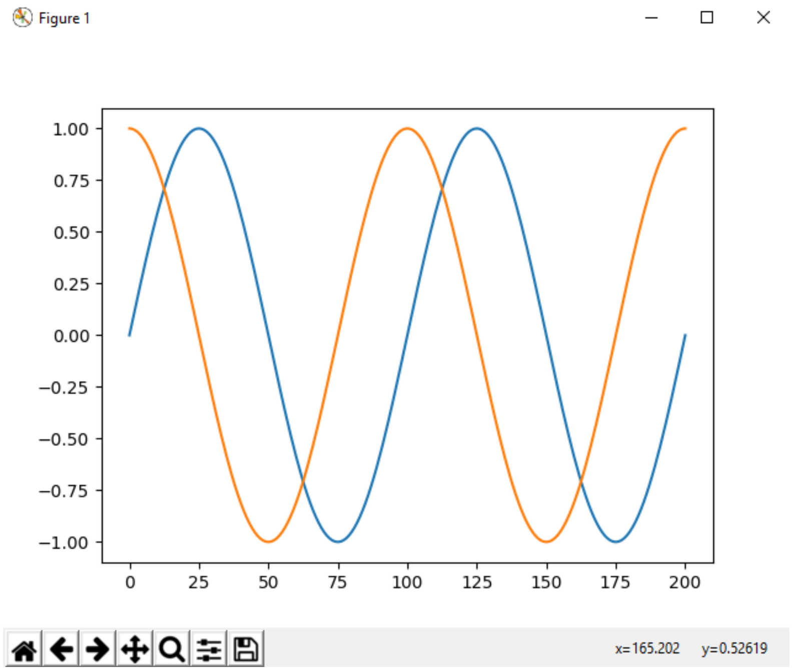
Sinusoidal Functions

LECTURE 3

Sine Function using Y-plot

Demo Program: sine.py

```
from pylab import *
sin_x = []
cos_x = []
# use range function to create sample
for x in range(-100, 101, 1):
    z = float(x/100 * 2 * math.pi)
    sin_x.append(sin(z))
    cos_x.append(cos(z))
plot(sin_x)
plot(cos_x)
show()
```



MATLAB-like API (pylab)

- The easiest way to get started with plotting using matplotlib is often to use the MATLAB-like API provided by matplotlib.
- It is designed to be compatible with MATLAB's plotting functions, so it is easy to get started with if you are familiar with MATLAB.

Matlab-Like pylab X-Y Plot

Demo Program: sine2.py

```
from pylab import *

x = linspace(-math.pi*1, math.pi*1, 100)
y = sin(x)
figure()             # create a figure (Container)
plot(x, y, 'r')      # create x-y plot using red dot
xlabel('x')          # label x-axis
ylabel('y')          # label y-axis
title('x-y-plot for Since')     #
show()
```

Linear Space
From $-\pi$ to π, 100 samples

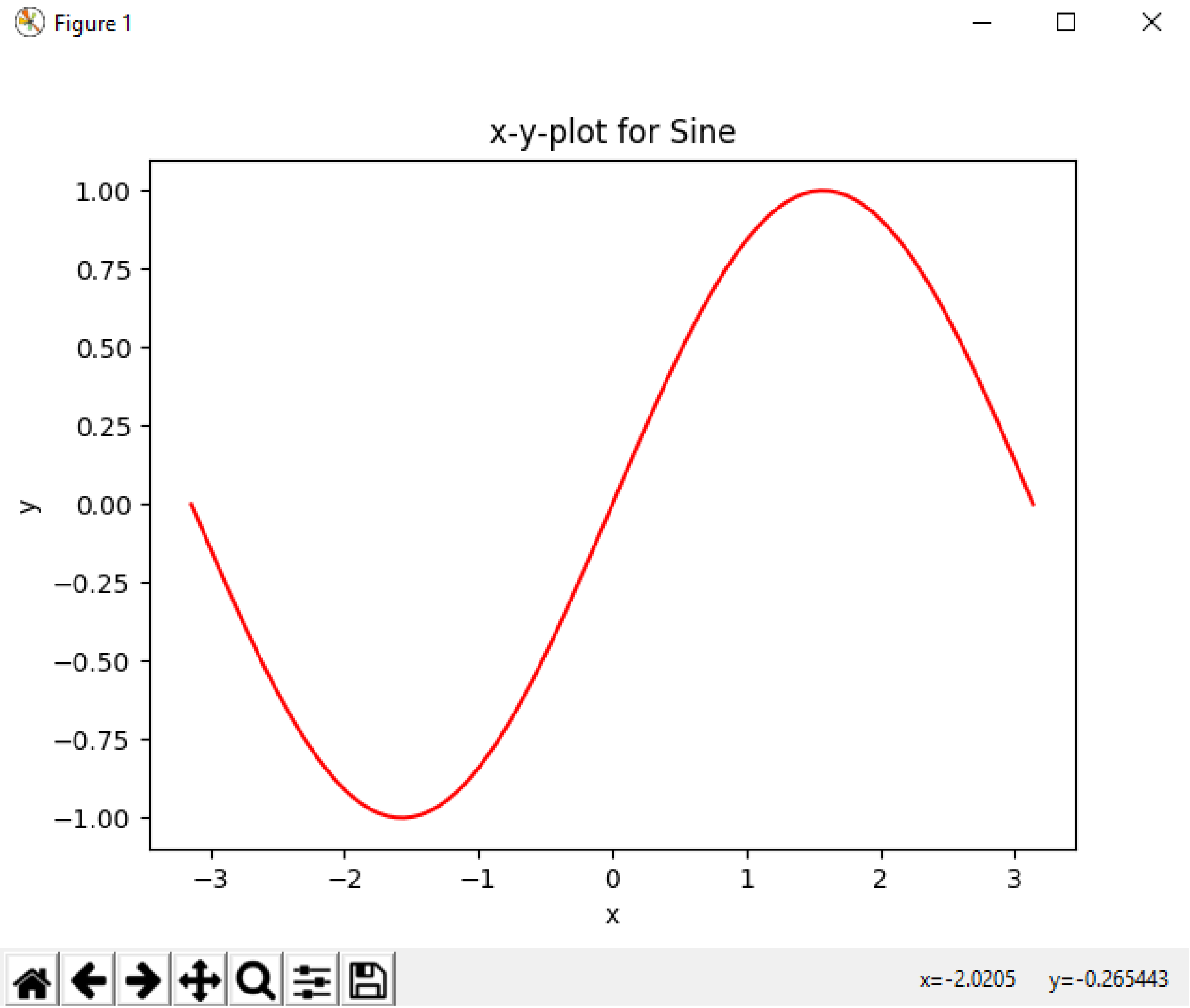


Figure 1
x-y-plot for Sine
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
y
−3
−2
−1
0
1
2
3
x
x=-2.0205 y=-0.265443

Sine X-Y Plot

Figure Style Design

(Grid/Axis Line)

LECTURE 4

grid(b=None, which='major', axis='both', **kwargs)

- Turn the axes grids **on** or **off**.
- Set the axes grids on or off; **b** is a boolean. (For MATLAB compatibility, b may also be a string, ‘on’ or ‘off’.)
- If **b** is None and len(kwargs)==0, toggle the grid state. If kwargs are supplied, it is assumed that you want a grid and b is thus set to True.
- which can be ‘**major**’ (default), ‘minor’, or ‘both’ to control whether major tick grids, minor tick grids, or both are affected.
- **axis** can be ‘both’ (default), ‘x’, or ‘y’ to control which set of gridlines are drawn.
- **grid**(color='r', linestyle='-', linewidth=2)

https://matplotlib.org/api/_as_gen/matplotlib.pyplot.html?highlight=matplotlib%20api

axhline(y=0, xmin=0, xmax=1, hold=None, **kwargs)

Parameters:	• **y** : scalar, optional, default: 0 y position in data coordinates of the horizontal line. • **xmin** : scalar, optional, default: 0 Should be between 0 and 1, 0 being the far left of the plot, 1 the far right of the plot. • **xmax** : scalar, optional, default: 1 Should be between 0 and 1, 0 being the far left of the plot, 1 the far right of the plot.
Returns:	Line2D

Polynomial

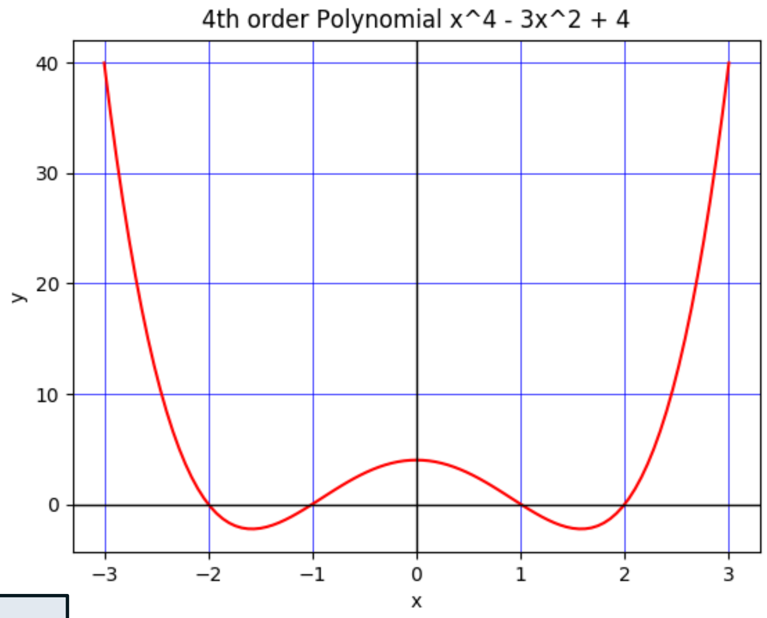
Demo Program: Polynomial1.py

```python
from pylab import *
def poly(x):
    return x**4-5*x**2 + 4

x = linspace(-3, 3, 101)
y = poly(x)
figure()            # create a figure
plot(x, y, 'r')     # create x-y plot using red dot
xlabel('x')         # label x-axis
ylabel('y')         # label y-axis
title('4th order Polynomial x^4 - 3x^2 + 4')    #
grid(linestyle='-', linewidth='0.5', color='blue')
axhline(0, color='black', lw=1.0)
axvline(0, color='black', lw=1.0)
show()
```

User-defined function

Horizontal and Vertical axis lines

Grid lines



Programmable Parameter List

PYLAB (MATLAB-LIKE DATA VISUALIZATION TOOL)

**kwargs : (Property Parameter List)

Valid kwargs are properties, with the exception of 'transform':

Property	Description
agg_filter	a filter function, which takes a (m, n, 3) float array and a dpi value, and returns a (m, n, 3) array
alpha	float (0.0 transparent through 1.0 opaque)
animated	bool
antialiased or aa	[True | False]
clip_box	a Bbox instance
clip_on	bool
clip_path	[(Path, Transform) | Patch | None]
color or c	any matplotlib color
contains	a callable function
dash_capstyle	['butt' | 'round' | 'projecting']
dash_joinstyle	['miter' | 'round' | 'bevel']
dashes	sequence of on/off ink in points
drawstyle	['default' | 'steps' | 'steps-pre' | 'steps-mid' | 'steps-post']
figure	a Figure instance
fillstyle	['full' | 'left' | 'right' | 'bottom' | 'top' | 'none']

**kwargs :

Valid kwargs are properties, with the exception of ‘transform’:

Property	Description
gid	an id string
label	object
linestyle or ls	[‘solid’ | ‘dashed’, ‘dashdot’, ‘dotted’ | (offset, on-off-dash-seq) | '-' | '--' | '-.' | ':' | 'None' | ' ' | '']
linewidth or lw	float value in points
marker	A valid marker style
markeredgecolor or mec	any matplotlib color
markeredgewidth or mew	float value in points
markerfacecolor or mfc	any matplotlib color
markerfacecoloralt or mfcalt	any matplotlib color
markersize or ms	float
markevery	[None | int | length-2 tuple of int | slice | list/array of int | float | length-2 tuple of float]

**kwargs :

Valid kwargs are properties, with the exception of ‘transform’:

Property	Description
path_effects	AbstractPathEffect
picker	float distance in points or callable pick function fn(artist,event)
pickradius	float distance in points
rasterized	bool or None
sketch_params	(scale: float, length: float, randomness: float)
snap	bool or None
solid_capstyle	[‘butt’ | ‘round’ | ‘projecting’]
solid_joinstyle	[‘miter’ | ‘round’ | ‘bevel’]
transform	a matplotlib.transforms.Transform instance
url	a url string
visible	bool
xdata	1D array
ydata	1D array
zorder	float

Finding Roots

LECTURE 5

Find Roots for Polynomial Using Scipy Optimize

- Using numerical Newton method.
- Not very straight forward to use.

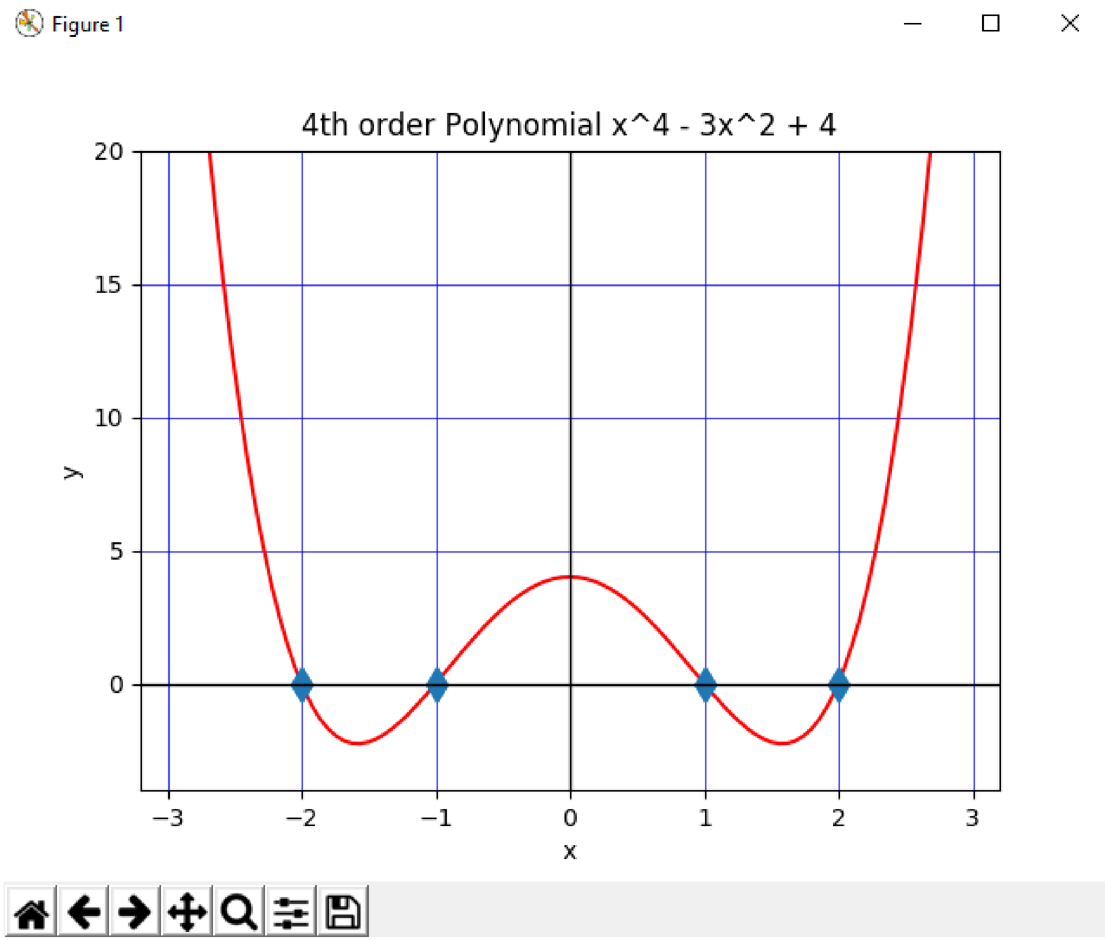
```python
from pylab import *
from scipy import optimize

def p4(x, a=1, b=0, c=-5, d=0, e=4):
    return a*x**4+b*x**3+c*x**2+d*x+e

# finding roots
x = linspace(-3, 3, 101)
y = p4(x, 1.0, 0.0, -5.0, 0.0, 4.0)
rx = optimize.root(p4, [-2, -1, 1, 2])

print(rx.x)
figure()                             # create a figure
xlim(xmin=-3.2, xmax=3.2)            # set x-axis range
ylim(ymin=-4, ymax=20)               # set y-axis range
plot(x, y, 'r')                      # create x-y plot using red dot
plot(rx.x, p4(rx.x), 'd', ms=10)
xlabel('x')                          # label x-axis
ylabel('y')                          # label y-axis
title('4th order Polynomial x^4 - 3x^2 + 4')    #
grid(linestyle='-', linewidth='0.5', color='blue')
axhline(0, color='black', lw=1.0)
axvline(0, color='black', lw=1.0)
show()
```

Find roots

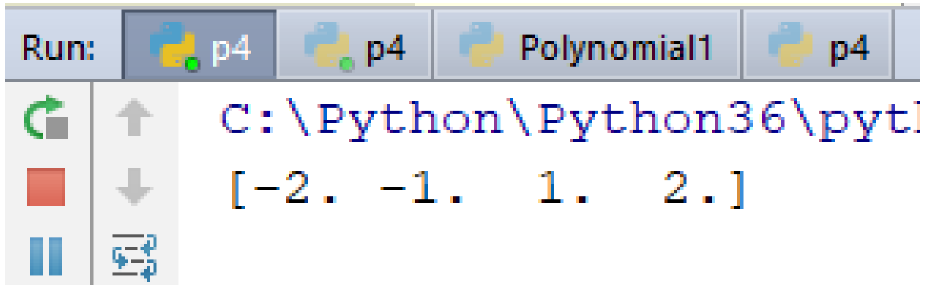
Setting the domain and range

plot roots and f(roots) as diamond

Numerical Roots by Scipy optimize.root



The roots rx.x:



Find Roots for Polynomial Using 1-D Numpy Polynomial: p4_2.py

- Easy if real roots.
- May not have robust optimization algorithm.
- p = numpy.poly1d(c) # is the coefficient list, used to create polynomial
- rx = numpy.roots(p)   # return a list of roots.

```python
from pylab import *
import numpy as np

# finding roots
x = linspace(-3, 3, 101)
c = [1, 0, -5, 0, 4]
p = np.poly1d(c)                   # create a numpy polynomial with c
y = p(x)                           # find the y value
rx = np.roots(p)                   # find the roots for the polynomial

print(rx)                          # print roots
figure()                           # create a figure
xlim(xmin=-3.2, xmax=3.2)          # set x-axis range
ylim(ymin=-4, ymax=20)             # set y-axis range
plot(x, y, 'r')                    # create x-y plot using red dot
plot(rx, p(rx), 'd', ms=10)        # plot root locations
xlabel('x')                        # label x-axis
ylabel('y')                        # label y-axis
title('Polynomial x^4 - 5x^2 + 4')    #
grid(linestyle='-', linewidth='0.5', color='blue')
axhline(0, color='black', lw=1.0)
axvline(0, color='black', lw=1.0)
show()
```

Learning Channel

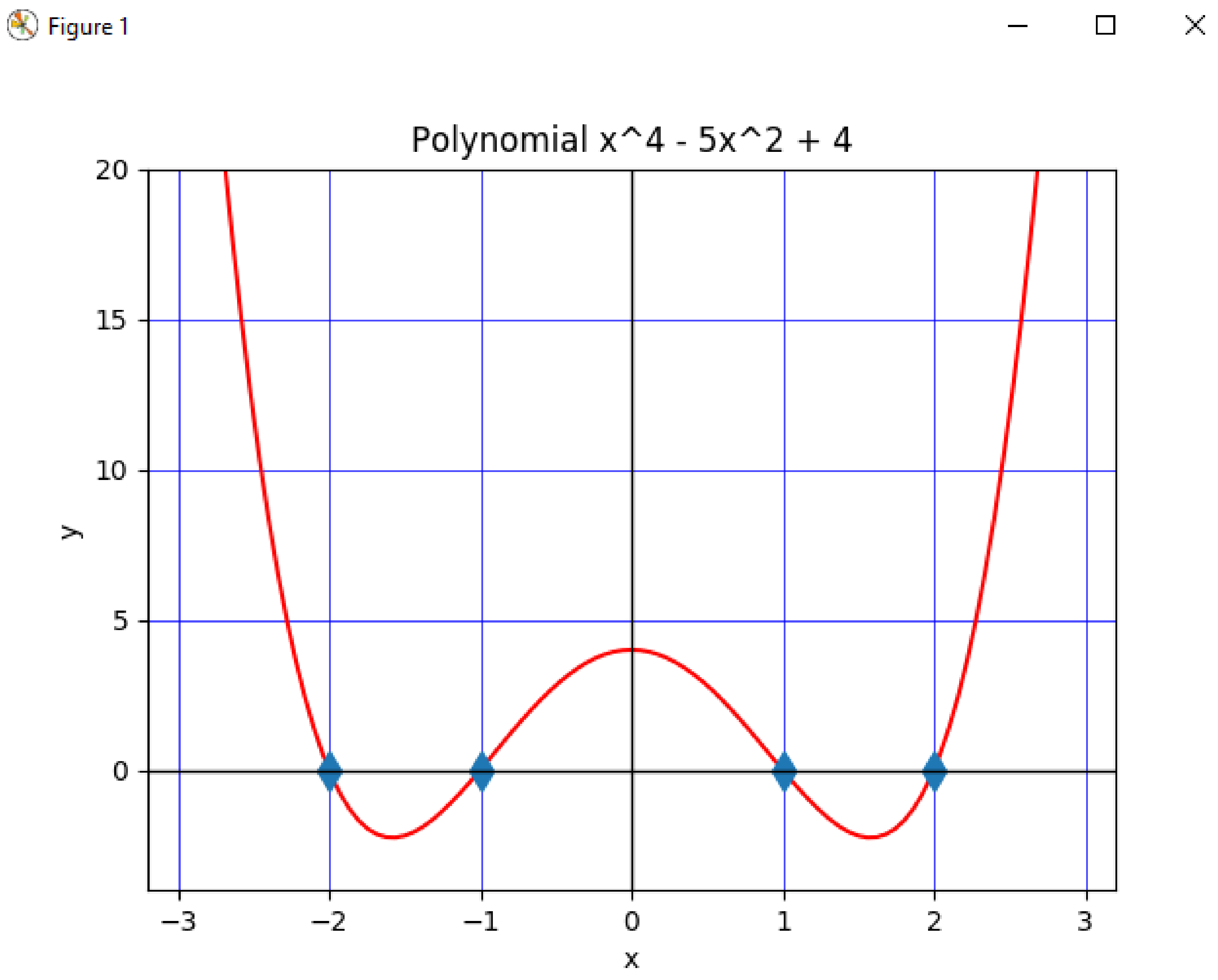


Figure 1
Polynomial x^4 - 5x^2 + 4
20
15
10
5
0
y
−3
−2
−1
0
1
2
3
x

Interactive Mode

LECTURE 6

Matplotlib from the Shell

- Using matplotlib in the shell, we see the objects produced

```
>>> from pylab import *
>>> plot([1,2,3,4])
[<matplotlib.lines.Line2D object at 0x01A3EF30>]
>>> show()
```

Prepare data Model

Update Screen

- Hangs until the figure is dismissed or `close()` issued
- **`show()`** clears the figure: issuing it again has no result
  - If you construct another figure, **`show()`** displays it without hanging

draw()

Clear the current figure and initialize a blank figure without hanging or displaying anything

Results of subsequent commands added to the figure

```
>>> draw()
>>> plot([1,2,3])
[<matplotlib.lines.Line2D object at 0x01C4B6B0>]
>>> plot([1,2,3],[0,1,2])
[<matplotlib.lines.Line2D object at 0x01D28330>]
```

- Shows 2 lines on the figure after **`show()`** is invoked

Use **`close()`** to dismiss the figure and clear it

`clf()` clears the figure (also deletes the white background) without dismissing it

Figures saved using **`savefig()`** in the shell by default are saved in

`C:\Python27`

- Override this by giving a full pathname

- **draw()**
- Clear the current figure and initialize a blank figure without hanging or displaying anything
- Results of subsequent commands added to the figure

```
>>> draw()
>>> plot([1,2,3])
[<matplotlib.lines.Line2D object at 0x01C4B6B0>]
>>> plot([1,2,3],[0,1,2])
[<matplotlib.lines.Line2D object at 0x01D28330>]
```

  - Shows 2 lines on the figure after `show()` is invoked
- Use `close()` to dismiss the figure and clear it
- `clf()` clears the figure (also deletes the white background) without dismissing it
- Figures saved using `savefig()` in the shell by default are saved in

  `C:\Python27`

  - Override this by giving a full pathname

Interactive Mode

- In interactive mode, objects are displayed as soon as they're created
- Use **`ion()`** to turn on interactive mode, **`ioff()`** to turn it off

Example

Import the pylab namespace, turn on interactive mode and plot a line

```
>>> from pylab import *
>>> ion()
>>> plot([1,2,3,4])
[<matplotlib.lines.Line2D object at 0x02694A10>]
```

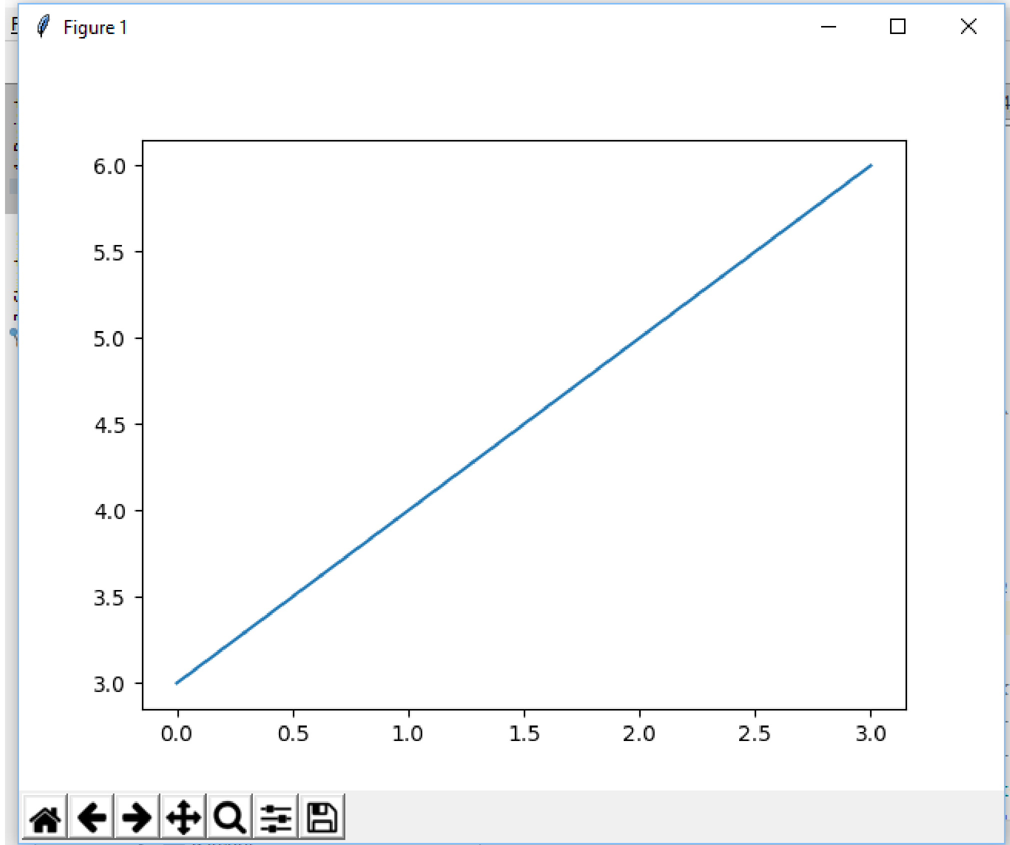
- A figure appears with this line on it
- The command line doesn't hang

- Plot another line

```
>>> plot([4,3,2,1])
[<matplotlib.lines.Line2D object at 0x00D68C30>]
```

- This line is shown on the figure as soon as the command is issued

Numericals - [C:\Eric_Chou\Python Course\PreCalculus with Python Programming\PyDev\Numericals] - ...\Polynomials\poly4.py - PyCha



Python Console

```
>>> from pylab import *
Backend TkAgg is interactive backend. Turning interactive mode on.
>>> ion
<function ion at 0x0000022437E7DD08>
>>> plot([3, 4, 5, 6])
[<matplotlib.lines.Line2D object at 0x000002243AA11FD0>]

>>>
```

Interactive Mode

- Turn off interactive mode and plot another line

```
>>> ioff()
>>> plot([3,2.5,2,1.5])
[<matplotlib.lines.Line2D object at 0x00D09090>]
```

  - The figure remains unchanged—no new line

- Update the figure

```
>>> show()
```

  - The figure now has all 3 lines
  - The command line is hung until the figure is dismissed

Cheat Sheet

LECTURE 7

Python & Pylab Cheat Sheet

Running

`python3`	standard python shell.
`ipython3`	improved interactive shell.
`ipython3 --pylab`	ipython including pylab
`python3` *file.py*	run *file.py*
`python3 -i` *file.py*	run *file.py*, stay in interactive mode

To quit use `exit()` or [ctrl]+[d]

Getting Help

`help()`	interactive Help
`help(`*object*`)`	help for *object*
object?	ipython: help for *object*
object??	ipython: extended help for *object*
`%magic`	ipython: help on magic commands

Import Syntax, e.g. for π

`import numpy`	use: `numpy.pi`
`import numpy as np`	use: `np.pi`
`from numpy import pi`	use: `pi`
`from numpy import *`	use: `pi` (use sparingly)

Types

`i = 1`	Integer		
`f = 1.`	Float		
`c = 1+2j`	Complex	with this:	
`True/False`	Boolean	`c.real`	1.0
`'abc'`	String	`c.imag`	2.0
`"abc"`	String	`c.conjugate()`	1-2j

Operators

mathematics		comparison	
`+`	addition	`=`	assign
`-`	subtraction	`==`	equal
`*`	multiplication	`!=`	unequal
`i/i`	int division	`<`	less
`i/f`	float division	`<=`	less-equal
`**`	power	`>=`	greater-equal
`%`	modulo	`>`	greater

Basic Syntax

`raw_input('`*foo*`')`	read string from command-line
`class` *Foo*`(Object):` ...	class definition
`def` *bar*`(args):` ...	function/method definition
`if` *c*`:` ... `elif` *c*`:` ... `else:`	branching
`try:` ... `except` *Error*`:` ...	exception handling
`while` *cond*`:` ...	while loop
`for` *item* `in` *list*`:` ...	for loop
`[`*item* `for` *item* `in` *list*`]`	for loop, list notation

Useful tools

`pylint` *file.py*	static code checker
`pydoc` *file*	parse docstring to man-page
`python3 -m doctest` *file.py*	run examples in docstring
`python3 -m pdb` *file.py*	run in debugger

NumPy & Friends

The following import statement is assumed:
`from pylab import *`

General Math

`f`: float, `c`: complex:

`abs(c)`	absolute value of f or c
`sign(c)`	get sign of f or c
`fix(f)`	round towards 0
`floor(f)`	round towards − inf
`ceil(f)`	round towards + inf
`f.round(p)`	round f to p places
`angle(c)`	angle of complex number
`sin(c)`	sinus of argument
`arcsin(c)`	arcsin of argument
`cos, tan,...`	analogous

Defining Lists, Arrays, Matrices

`l`: list, `a`: array:

`[[1,2],[3,4,5]]`	basic list
`array([[1,2],[3,4]])`	array from "rectangular" list
`matrix([[1,2],[3,4]])`	matrix from 2d-list
`range(min, max, step)`	integers in [min, max)
`list(range(...))`	list from range()
`arange(min, max, step)`	integer array in [min, max)
`frange(min, max, step)`	float array in [min, max]
`linspace(min, max, num)`	num samples in [min, max]
`meshgrid(x,y)`	create coord-matrices
`zeros, ones, eye`	generate special arrays

Element Access

`l[row][col]`	list: basic access
`l[min:max]`	list: range access [min,max)
`a[row,col]` or `a[row][col]`	array: basic access
`a[min:max,min:max]`	array: range access [min,max)
`a[`*list*`]`	array: select indices in *list*
`a[np.where(`*cond*`)]`	array: select where *cond* true

List/Array Properties

`len(l)`	size of first dim
`a.size`	total number of entries
`a.ndim`	number of dimensions
`a.shape`	size along dimensions
`ravel(l)` or `a.ravel()`	convert to 1-dim
`a.flat`	iterate all entries

Matrix Operations

`a`: array, `M`: matrix:

`a*a`	element-wise product
`dot(a,a)` or `M*M`	dot product
`cross(a,a)`	cross product
`inv(a)` or `M.I`	inverted matrix
`transpose(a)` or `M.T`	transposed matrix
`det(a)`	calculate determinate

Statistics

`sum(l,d)` or `a.sum(d)`	sum elements along d
`mean(l,d)` or `a.mean(d)`	mean along d
`std(l,d)` or `a.std(d)`	standard deviation along d
`min(l,d)` or `a.min(d)`	minima along d
`max(l,d)` or `a.max(d)`	maxima along d

Misc functions

`loadtxt(`*file*`)`	read values from *file*
`polyval(coeff,xvals)`	evaluate polynomial at xvals
`roots(coeff)`	find roots of polynomial
`map(`*func*`,`*list*`)`	apply func on each element of list

Plotting

Plot Types

`plot(xvals, yvals, 'g+')`	mark 3 points with green +
`errorbar()`	like `plot` with error bars
`semilogx(), semilogx()`	like `plot`, semi-log axis
`loglog()`	double logarithmic plot
`polar(phi_vals, rvals)`	plot in polar coordinates
`hist(vals, n_bins)`	create histogram from values
`bar(low_edge, vals, width)`	create bar-plot
`contour(xvals,yvals,zvals)`	create contour-plot

Pylab Plotting Equivalences

`figure()`	`fig = figure()`
	`ax = axes()`
`subplot(2,1,1)`	`ax = fig.add_subplot(2,1,1)`
`plot()`	`ax.plot()`
`errorbar()`	`ax.errorbar()`
`semilogx, ...`	analogous
`polar()`	`axes(polar=True)` and `ax.plot()`
`axis()`	`ax.set_xlim(), ax.set_ylim()`
`grid()`	`ax.grid()`
`title()`	`ax.set_title()`
`xlabel()`	`ax.set_xlabel()`
`legend()`	`ax.legend()`
`colorbar()`	`fig.colorbar(plot)`

Plotting 3D

`from mpl_toolkits.mplot3d import Axes3D`

`ax = fig.add_subplot(...,projection='3d')`	
or `ax = Axes3D(fig)`	create 3d-axes object
`ax.plot(xvals, yvals, zvals)`	normal plot in 3d
`ax.plot_wireframe`	wire mesh
`ax.plot_surface`	colored surface