

Visualization and optimization

- Matplotlib
- Jupyter
- `scipy.optimize.minimize`



Matplotlib is a Python 2D plotting library which produces publication quality figures in a variety of hardcopy formats and interactive environments across platforms. Matplotlib can be used in Python scripts, the Python and IPython shells, the Jupyter notebook, web application servers, and four graphical user interface toolkits.

Matplotlib tries to make easy things easy and hard things possible. You can generate plots, histograms, power spectra, bar charts, errorcharts, scatterplots, etc., with just a few lines of code. For simple plotting the pyplot module provides a MATLAB-like interface, particularly when combined with IPython. For the power user, you have full control of line styles, font properties, axes properties, etc, via an object oriented interface or via a set of functions familiar to MATLAB users.

pip install matplotlib

matplotlib.org

Plot

pyplot module \approx MATLAB-like plotting framework

matplotlib-simple.py

```
import matplotlib.pyplot as plt  
  
plt.plot([1, 2, 3], [5, 2, 7], 'bo:')  
  
plt.show()
```

add plot
to figure

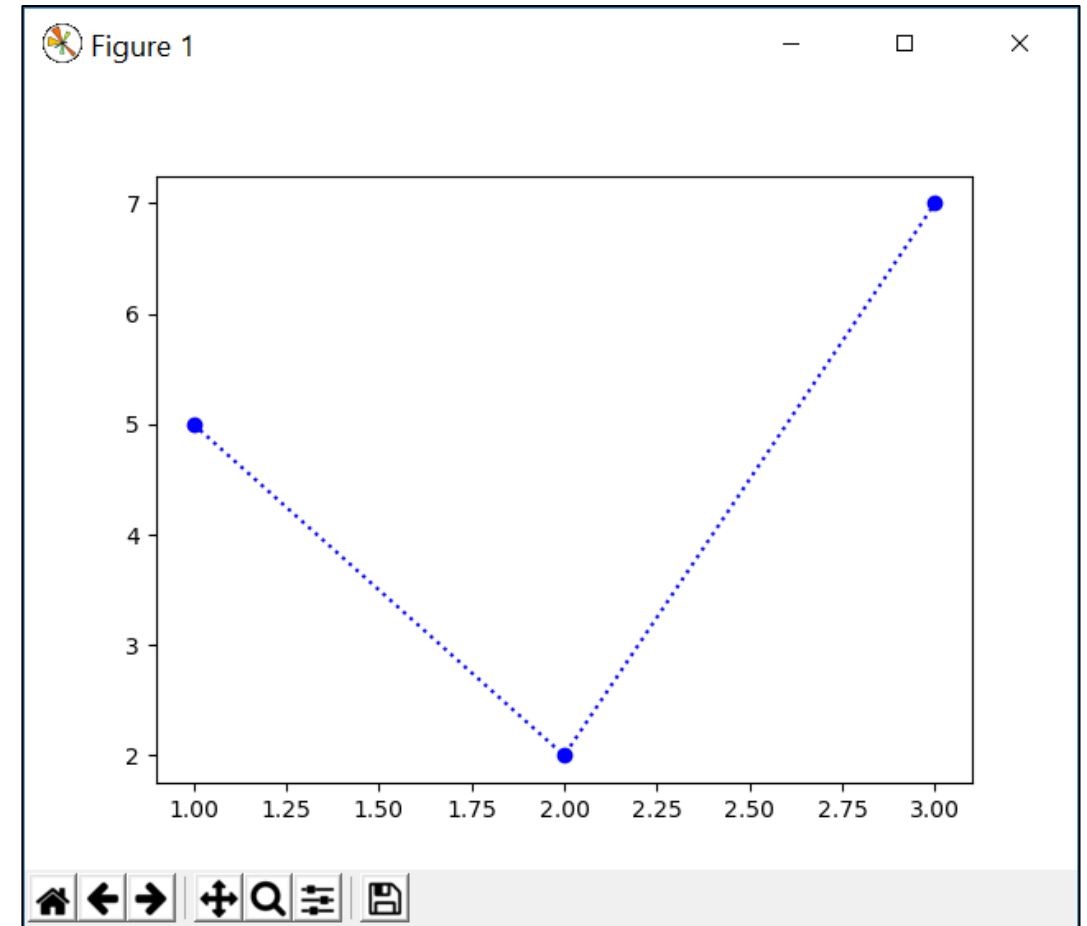
figure is first shown
when show is called

x coordinates

y coordinates

format
string

Colors	Line styles	Marker styles			
b	—	.	2	+	+
g	--	,	3	x	x
r	-.	o	4	D	◆
c	:	v	s	d	◆
m		^	p		
y		<	*	*	—
k		>	h	•	
w		1	H	•	



save current view as picture
adjust margins
zoom rectangle
pan and zoom
navigate view history
reset view

Plot – some keyword arguments

matplotlib-plot.py

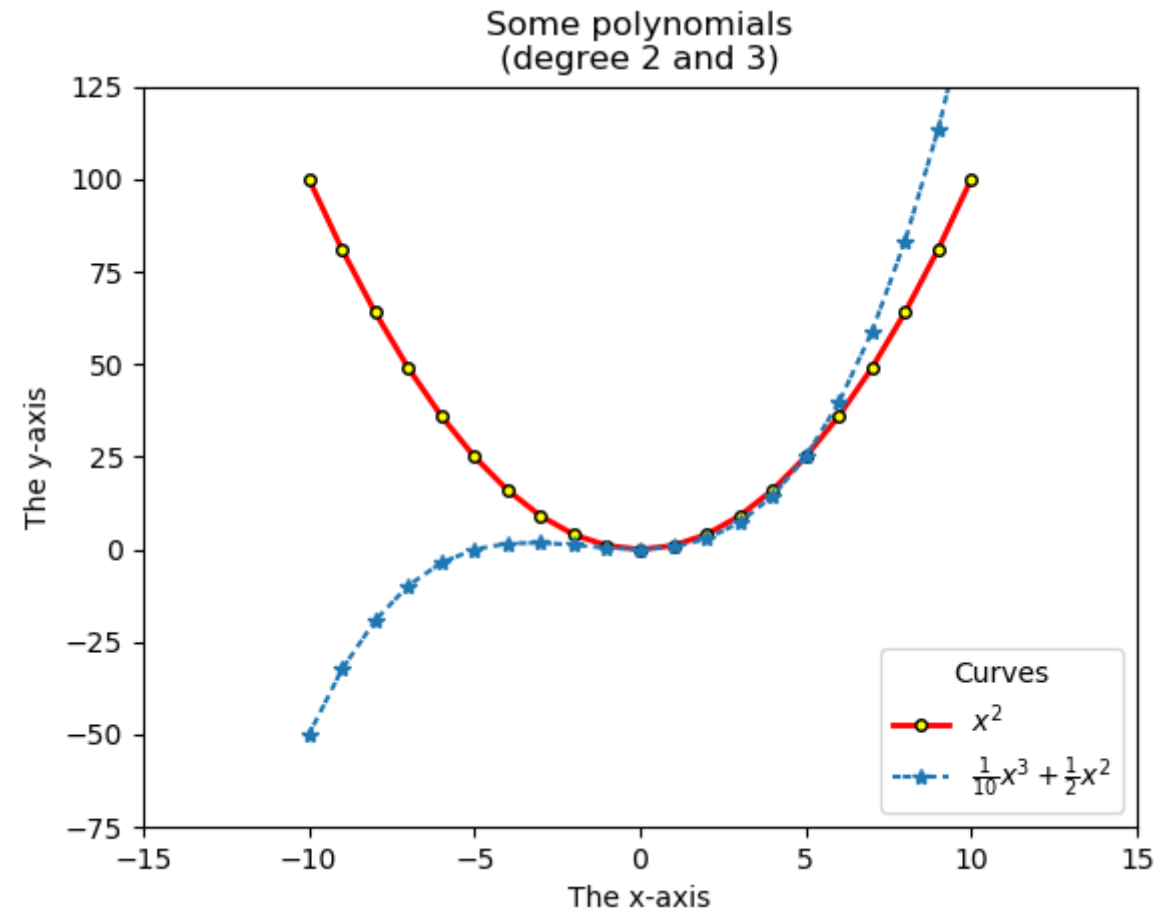
```
import matplotlib.pyplot as plt

X = range(-10, 11)
Y1 = [x ** 2 for x in X]
Y2 = [x ** 3 / 10 + x ** 2 / 2 for x in X]

plt.plot(X, Y1, color='red', label='$x^2$',
         linestyle='-', linewidth=2,
         marker='o', markersize=4,
         markeredgewidth=1,
         markeredgecolor='black',
         markerfacecolor='yellow')

plt.plot(X, Y2, '*', dashes=(2, 0.5, 2, 1.5),
         label=r'$\frac{1}{10}x^3+\frac{1}{2}x^2$')
plt.xlim(-15, 15)
plt.ylim(-75, 125)
plt.title('Some polynomials\n(degree 2 and 3)')
plt.xlabel('The x-axis')
plt.ylabel('The y-axis')
plt.legend(title='Curves')

plt.show() # finally show figure
```



matplotlib.org/api/ as gen/matplotlib.pyplot.plot.html

Colors: matplotlib.org/gallery/color/named_colors.html

Scatter (points with individual size and color)

colorbar
(of most recently
used colormap)

matplotlib-scatter.py

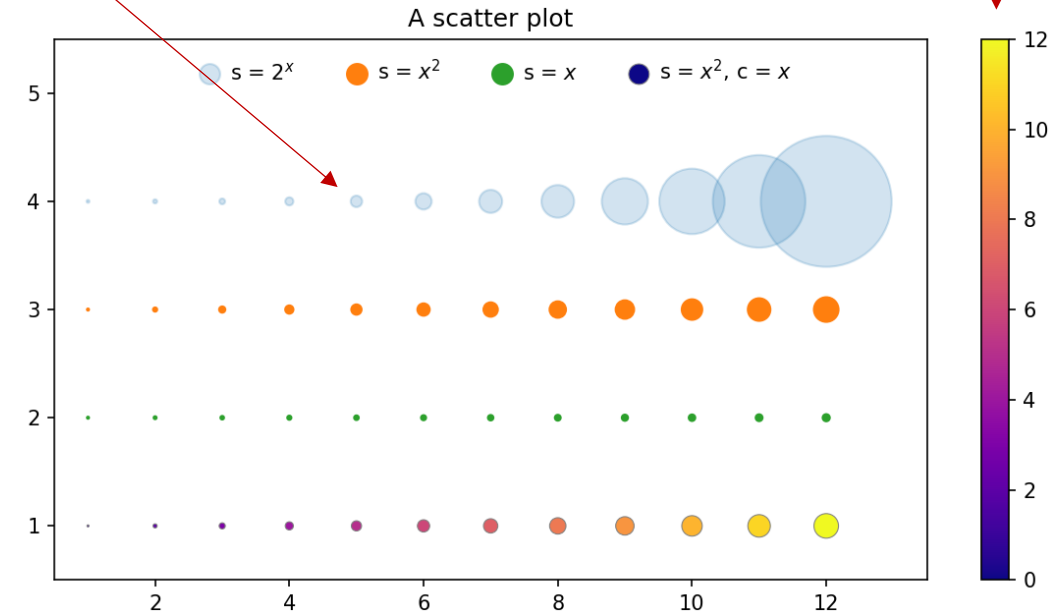
```
import matplotlib.pyplot as plt

n = 13
X = range(n)
S = [x ** 2 for x in X]
E = [2 ** x for x in X]

plt.scatter(X, [4] * n, s=E, label='s =  $2^x$ ', alpha=.2)
plt.scatter(X, [3] * n, s=S, label='s =  $x^2$ ')
plt.scatter(X, [2] * n, s=X, label='s =  $x$ ')
plt.scatter(X, [1] * n, s=S, c=X, cmap='plasma',
            label='s =  $x^2$ , c =  $x$ ',
            edgecolors='gray', linewidth=0.5)
plt.colorbar()

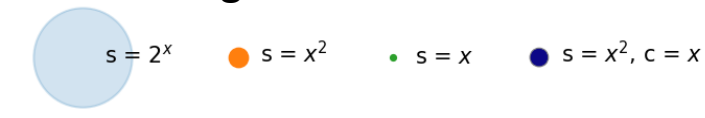
plt.ylim(0.5, 5.5)
plt.xlim(0.5, 13.5)
plt.title('A scatter plot')
legend = plt.legend(loc='upper center', frameon=False,
                   ncol=4, handletextpad=0)
for handle in legend.legend_handles:
    handle.set_sizes([100]) # fix marker size in legend
plt.show()
```

transparency



colormap (predefined)
color of each point
size \approx area of each point
point boundary width
point boundary color

Legend without fix



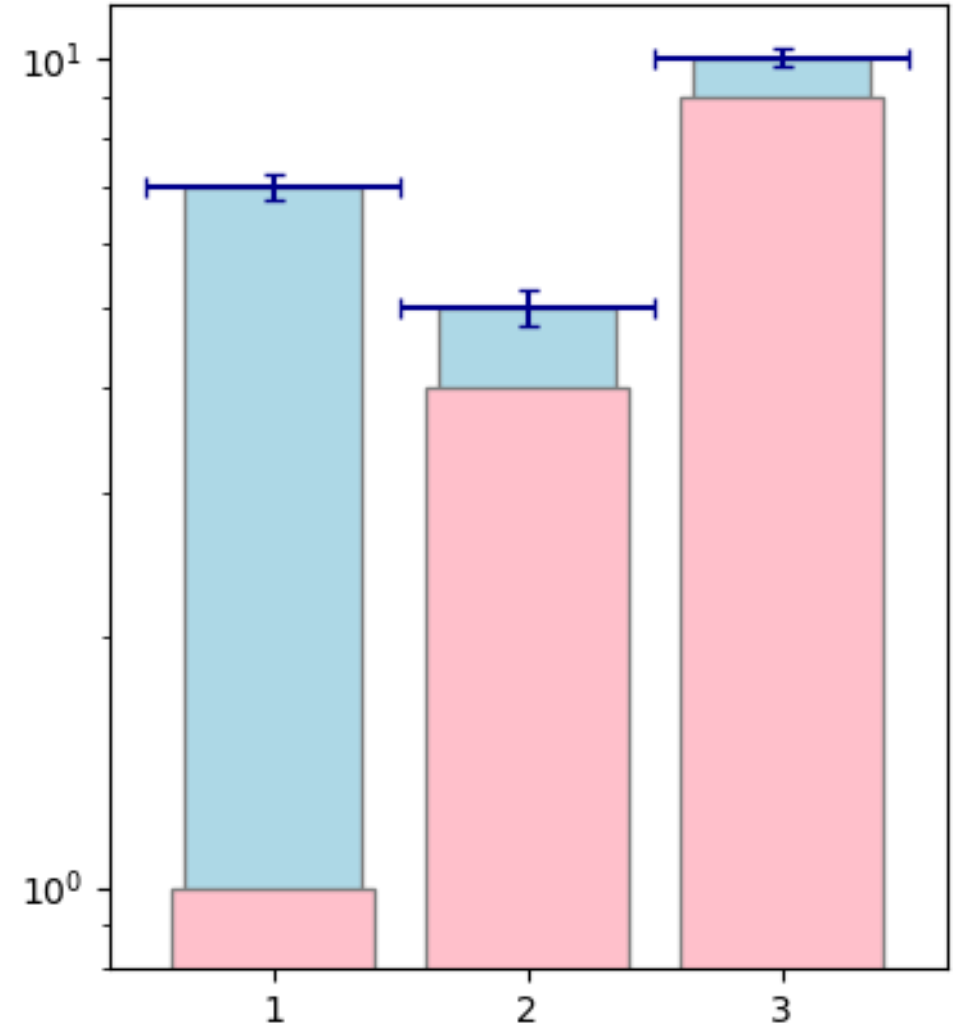
manual placement of legend box (default automatic); remove frame; place legends in 4 columns (default 1); reduce space between marks and label

matplotlib.org/api/_as_gen/matplotlib.pyplot.scatter.html
matplotlib.org/tutorials/colors/colormaps.html

Bars

matplotlib-bars.py

```
import matplotlib.pyplot as plt
x = [1, 2, 3]
y = [7, 5, 10]
plt.bar(x, y,
        color='lightblue', # bar background color
        linewidth=1,       # bar boundary width
        edgecolor='gray',  # bar boundary color
        tick_label=x,      # ticks on x-axis
        width=0.7,         # width, default 0.8
        yerr=0.25,         # Error bar: y length
        xerr=0.5,          # x length
        capsize=3,         # capsize in points
        ecolor='darkblue', # error bar color
        log=True)          # y-axis log scale
plt.bar(x, [v**2 for v in x],
        color='pink',
        linewidth=1,
        edgecolor='gray')
plt.show()
```



Histogram

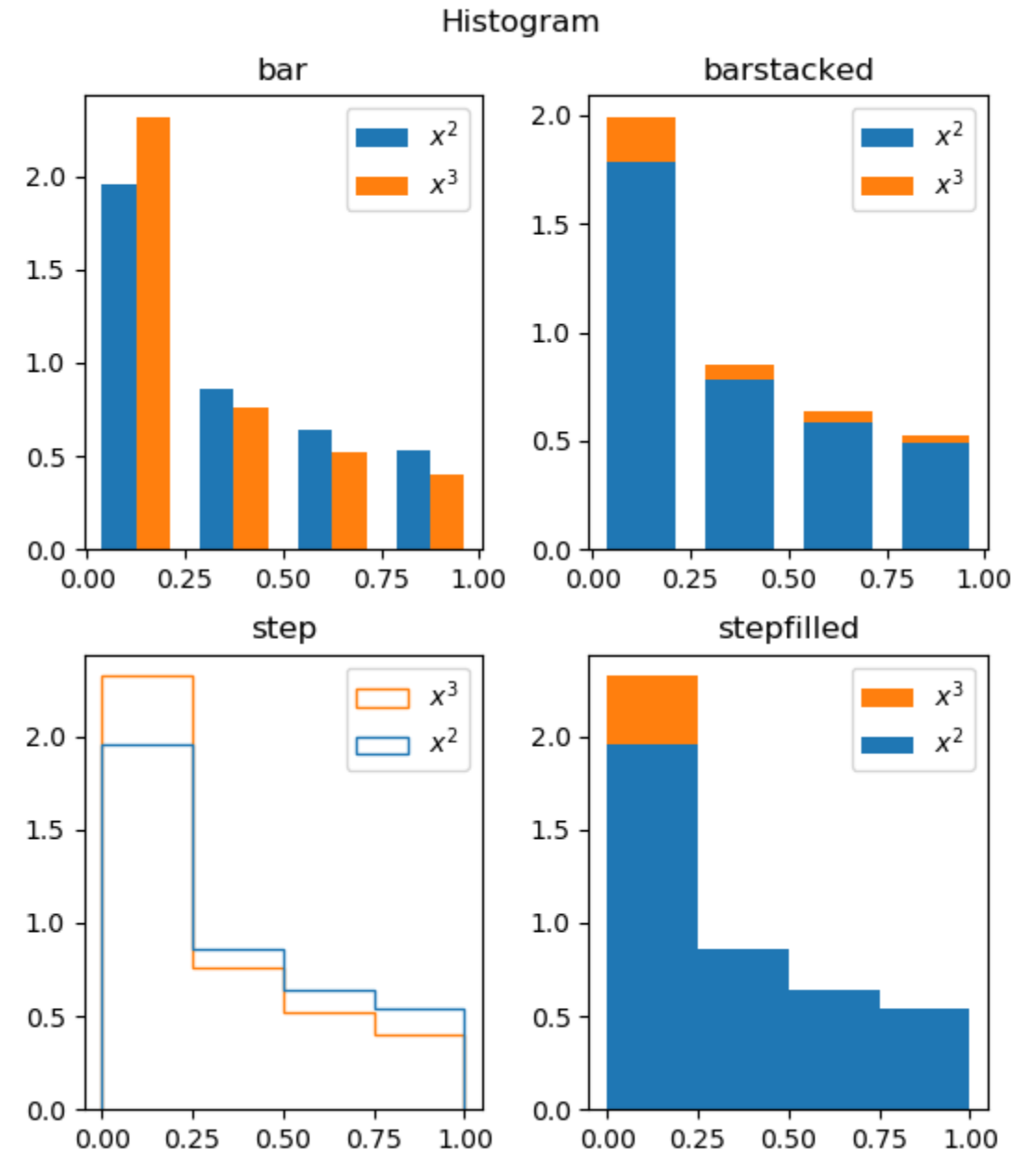
matplotlib-histogram.py

```
import matplotlib.pyplot as plt
from random import random

values1 = [random()*2 for _ in range(1000)]
values2 = [random()*3 for _ in range(100)]
bins = [0.0, 0.25, 0.5, 0.75, 1.0]

for i, ht in enumerate(
    ['bar', 'barstacked', 'step', 'stepfilled'],
    start=1):
    plt.subplot(2, 2, i) # start new plot
    plt.hist([values1, values2], # data sets
             bins, # bucket boundaries
             histtype=ht, # default ht='bar'
             rwidth=0.7, # fraction of bucket width
             label=['$x^2$', '$x^3$'], # labels
             density=True) # norm. prob. density
    plt.title(ht) # plot title
    plt.xticks(bins) # ticks on x-axis
    plt.legend()

plt.suptitle('Histogram') # figure title
plt.show()
```

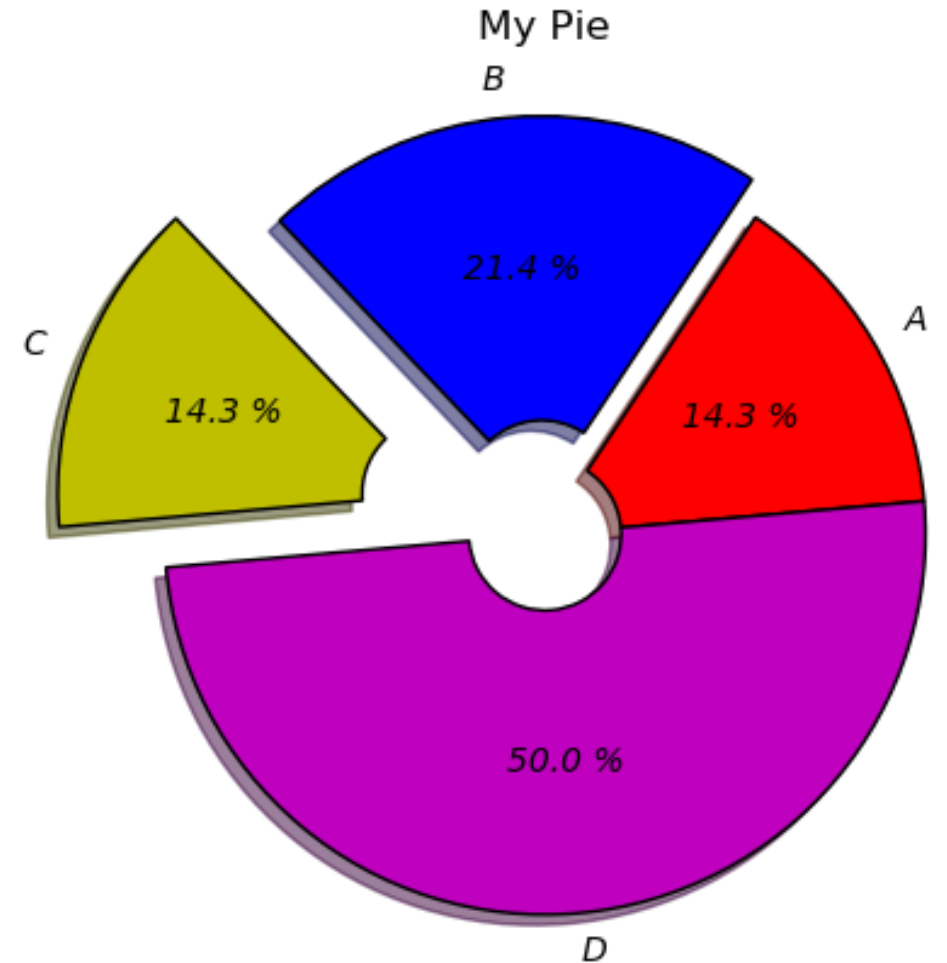


Pie

matplotlib-pie.py

```
import matplotlib.pyplot as plt
plt.title('My Pie')
plt.pie([2, 3, 2, 7],          # relative wedge sizes
        labels=['A', 'B', 'C', 'D'],
        colors=['r', 'b', 'y', 'm'],
        explode=(0, 0.1, 0.3, 0), # radius fraction
        startangle=5,           # angle above horizontal
        counterclock=True,      # default True
        rotatelabels=False,     # default False
        shadow=True,           # default False
        textprops=dict(        # text properties, dict
            color='black',      # text color
            style='italic'),     # text style
        wedgeprops=dict(       # wedge properties, dict
            width=0.8,          # width (missing center)
            linewidth=1,        # wedge boundary width
            edgecolor='black'),  # boundary color
        autopct='%1.1f %%')    # percent formatting

plt.show()
```



Customizing Pie shadows

- Need to do it manually on each pie using `matplotlib.patches.Shadow`

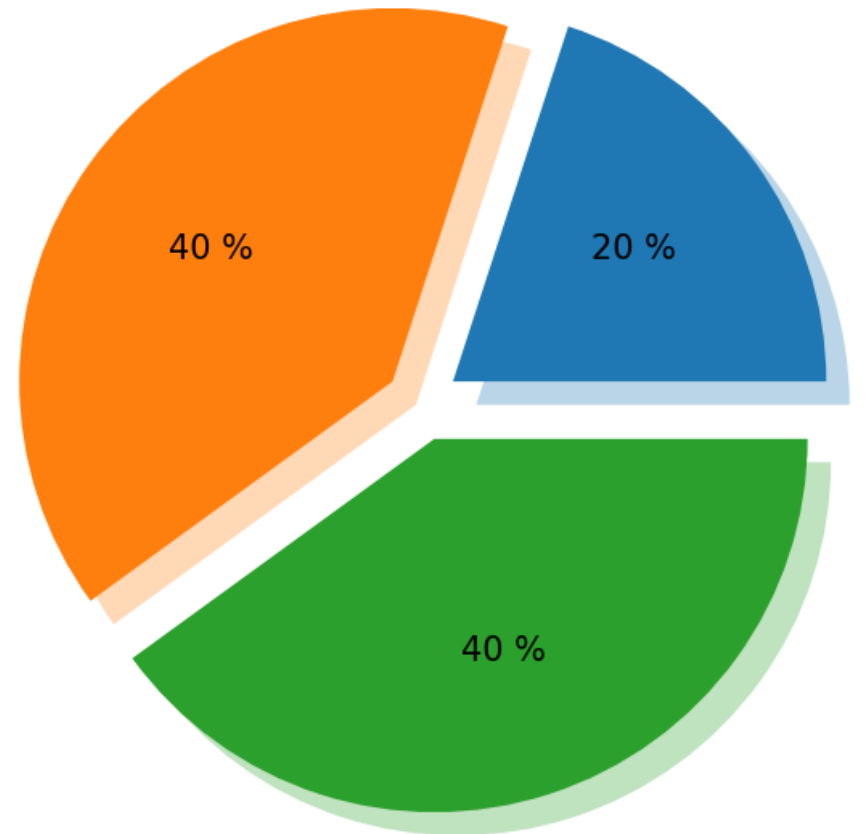
`matplotlib-pie-shadow.py`

```
import matplotlib.pyplot as plt
from matplotlib.patches import Shadow

patches, texts, autotexts = plt.pie(
    [1, 2, 2],
    explode=(0.1, 0.1, 0.1),
    autopct='%1.0f %%'
)

for pie in patches:
    pie_shadow = Shadow(
        pie, 0.03, -0.03, # patch, x-offset, y-offset
        alpha=0.3,         # shadow transparency
        edgecolor=None,    # shadow edge color
        facecolor=pie._facecolor # shadow fill color
    )
    plt.gca().add_patch(pie_shadow)

plt.show()
```



Stackplot

matplotlib-stackplot.py

```
import matplotlib.pyplot as plt
x = [1, 2, 3, 4]
y1 = [1, 2, 3, 4]
y2 = [2, 3, 1, 4]
y3 = [2, 4, 1, 3]

plt.style.use('dark_background')
for i, base in enumerate(
    ['zero', 'sym', 'wiggle', 'weighted_wiggle'],
    start=1):
    plt.subplot(4, 1, i)
    plt.stackplot(x, y1, y2, y3,
                  colors=['r', 'g', 'b'],
                  labels=['Red', 'Green', 'Blue'],
                  baseline=base)

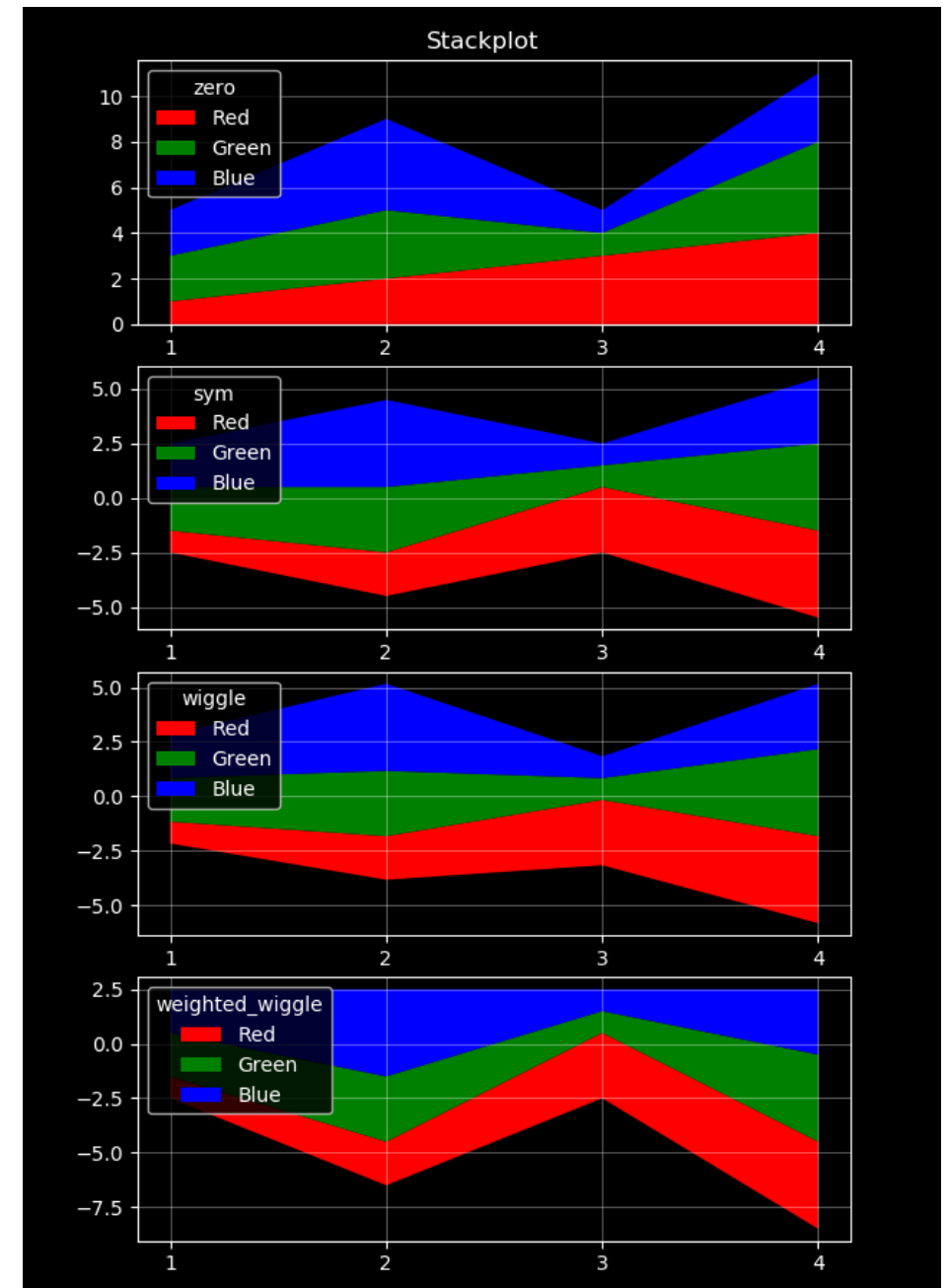
    plt.grid(axis='both', # 'x', 'y', or 'both'
            linewidth=0.5, linestyle='-', alpha=0.5)
    plt.legend(title=base, loc='upper left')
    plt.xticks(x) # a tick for each value in x

plt.suptitle('Stackplot')
plt.show()
```

Stacked Graphs – Geometry & Aesthetics
Lee Byron & Martin Wattenberg, 2008

To list all available styles:

```
print(plt.style.available)
```



```

import matplotlib.pyplot as plt
from math import pi, sin

x_min, x_max, n = 0, 2 * pi, 100
x = [x_min + (x_max - x_min) * i / n for i in range(n + 1)]
y = [sin(v) for v in x]

ax1 = plt.subplot(2, 3, 1) # 2 rows, 3 columns
ax1.label_outer()         # removes x-axis labels
plt.xlim(-pi, 3 * pi)     # increase x-axis range
plt.plot(x, y, 'r-')
plt.title('Plot A')

ax2 = plt.subplot(2, 3, 2)
ax2.label_outer()         # removes x- and y-axis labels
plt.xlim(-2 * pi, 4 * pi) # increase x-axis range
plt.plot(x, y, 'g-')
plt.title('Plot B')

ax3 = plt.subplot(2, 3, 3, frameon=False) # remove frame
ax3.set_xticks([])                   # remove x-axis ticks & labels
ax3.set_yticks([])                   # remove y-axis ticks & labels
plt.plot(x, y, 'b--')
plt.title('No frame')

ax4 = plt.subplot(2, 3, 4, sharex=ax1) # share x-axis range
plt.ylim(-2, 2)                        # increase y-axis range
plt.plot(x, y, 'm-')
plt.title('Plot C')

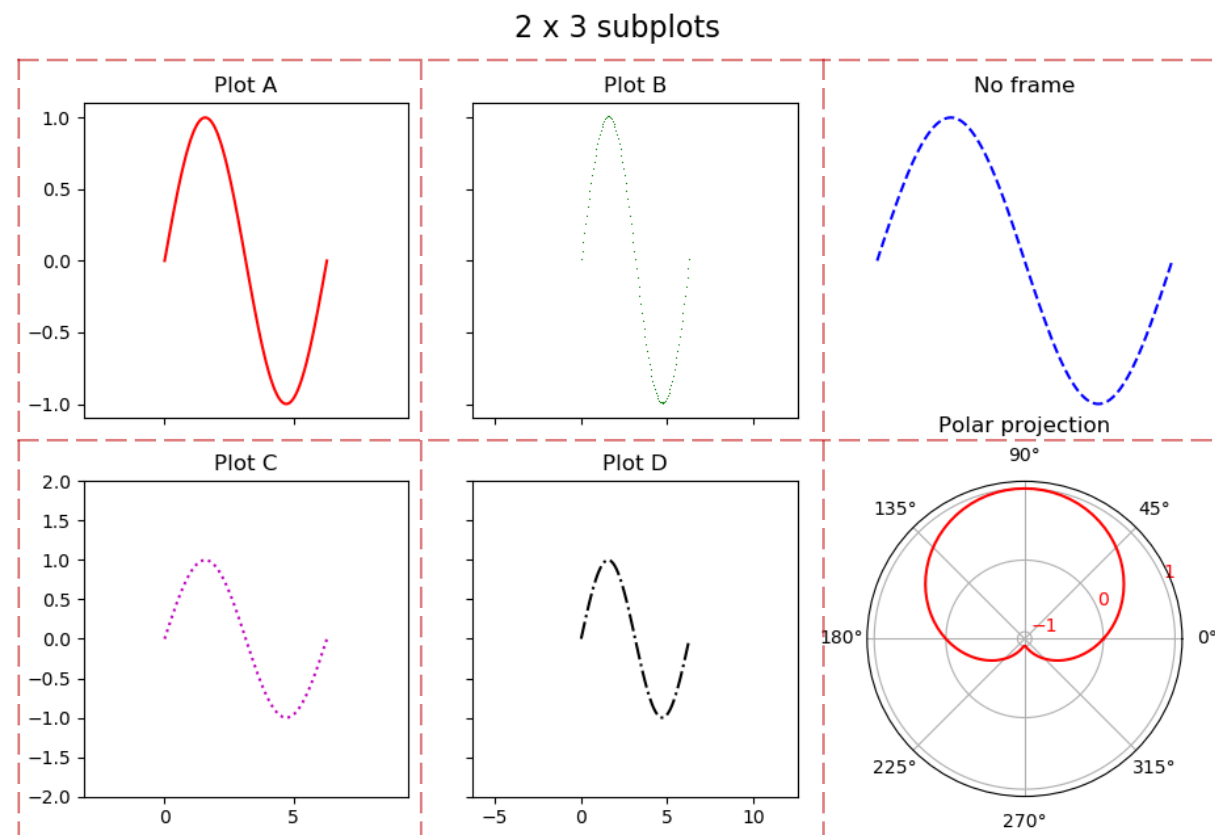
ax5 = plt.subplot(2, 3, 5, sharex=ax2, sharey=ax4) # share ranges
ax5.set_xticks(range(-5, 15, 5)) # specific x-ticks & x-labels
ax5.label_outer()                # removes y-axis labels
plt.plot(x, y, 'k-')
plt.title('Plot D')

ax6 = plt.subplot(2, 3, 6, projection='polar') # polar projection
ax6.set_yticks([-1, 0, 1])                    # y-labels
ax6.tick_params(axis='y', labelcolor='red')    # color of y-labels
plt.plot(x, y, 'r')
plt.title('Polar projection\n') # \n to avoid overlap with 90°
plt.suptitle('2 x 3 subplots', fontsize=16)
plt.show()

```

Subplot

(2 rows, 3 columns)



- Subplots are numbered 1..6 row-by-row, starting top-left
- subplot returns an `axes` to access the plot in the figure

Subplots

matplotlib-subplots.py

```
import matplotlib.pyplot as plt
from math import pi, sin, cos

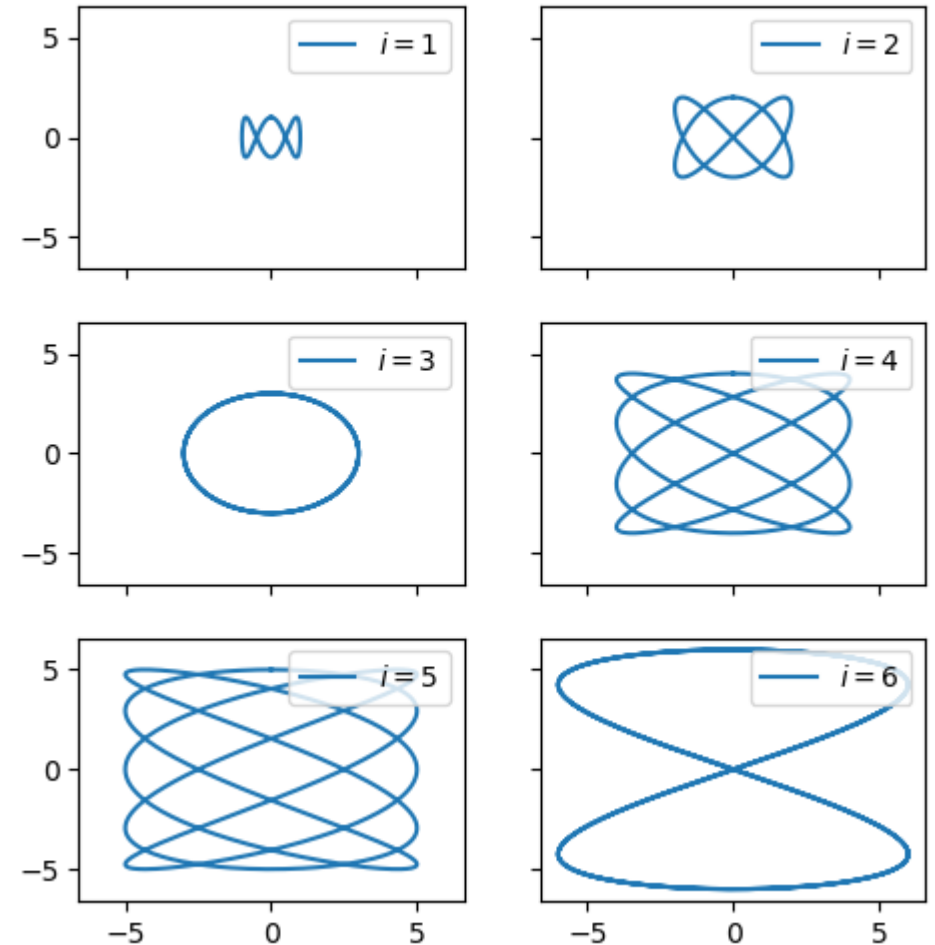
times = [2 * pi * t / 1000 for t in range(1001)]

fig, ((ax1, ax2), (ax3, ax4), (ax5, ax6)) = \
    plt.subplots(3, 2, sharex=True, sharey=True)

for i, ax in enumerate([ax1, ax2, ax3, ax4, ax5, ax6],
                        start=1):
    x = [i * sin(i * t) for t in times]
    y = [i * cos(3 * t) for t in times]
    ax.plot(x, y, label=f'$i = {i}$') # plot to axes
    ax.legend(loc='upper right')      # axes legend
fig.suptitle('subplots', fontsize=16) # figure title
plt.show()
```

create 6 axes in 3 rows with 2 columns
share the x- and y-axis ranges (automatically
applies label_outer to created axes)
returns a pair (figure, axes)

subplots



matplotlib-subplot2grid.py

```
import matplotlib.pyplot as plt
import math

x_min, x_max, n = 0, 2 * math.pi, 20

x = [x_min + (x_max - x_min) * i / n
      for i in range(n + 1)]
y = [math.sin(v) for v in x]

plt.subplot2grid((5, 5), (0, 0),
                 rowspan=3, colspan=3)
plt.fill_between(x, 0.0, y,
                 alpha=0.25, color='r')
plt.plot(x, y, 'r-')
plt.title('Plot A')

plt.subplot2grid((5, 5), (0, 3),
                 rowspan=2, colspan=2)
plt.plot(x, y, 'g.')
plt.title('Plot B')

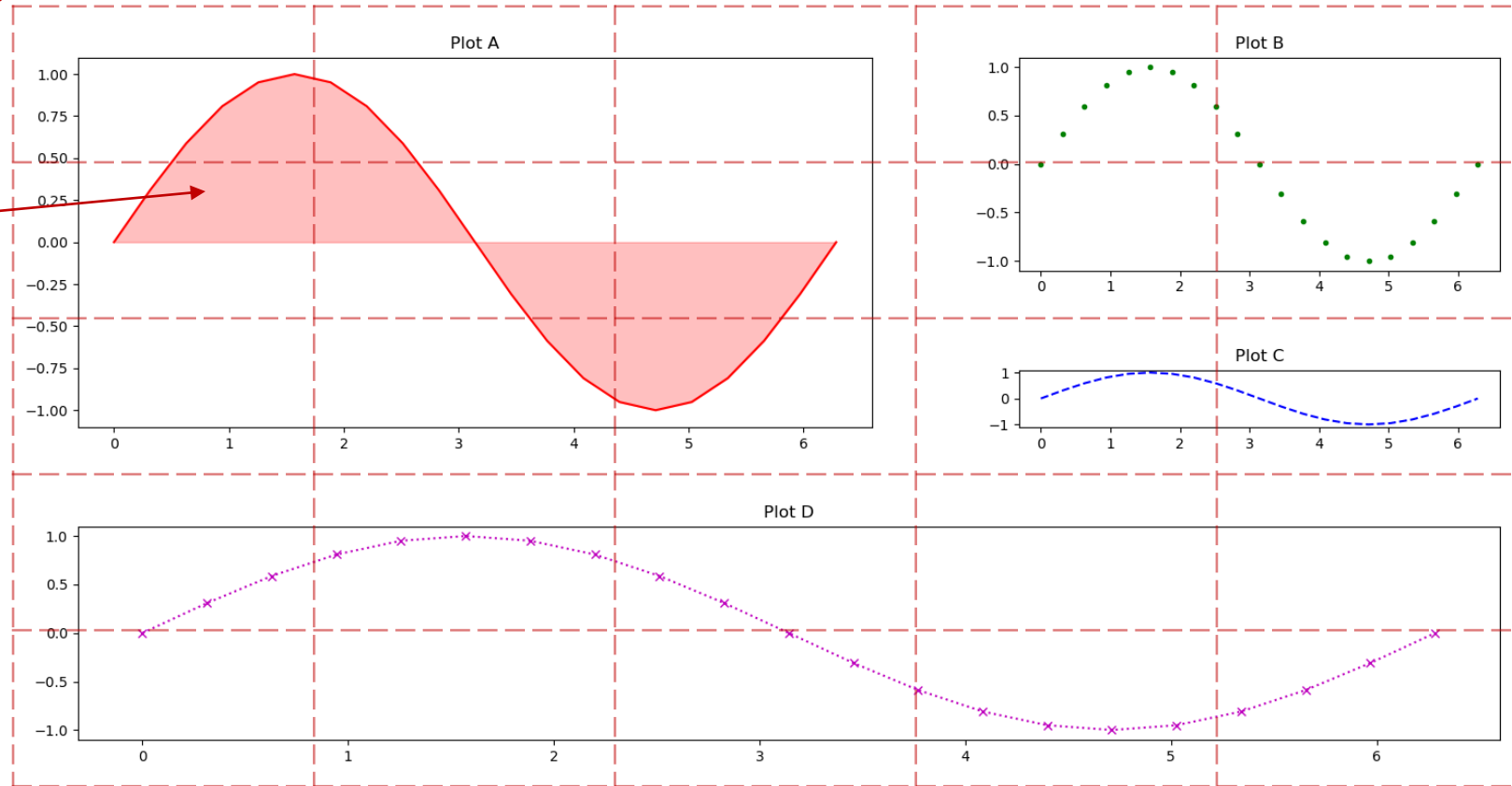
plt.subplot2grid((5, 5), (2, 3),
                 rowspan=1, colspan=2)
plt.plot(x, y, 'b--')
plt.title('Plot C')

plt.subplot2grid((5, 5), (3, 0),
                 rowspan=2, colspan=5)
plt.plot(x, y, 'mx:')
plt.title('Plot D')

plt.tight_layout() # adjust padding
plt.show()
```

subplot2grid (5 x 5)

upper left corner (row, column)



```

import matplotlib.pyplot as plt

x = [i / 10 for i in range(1, 101)]

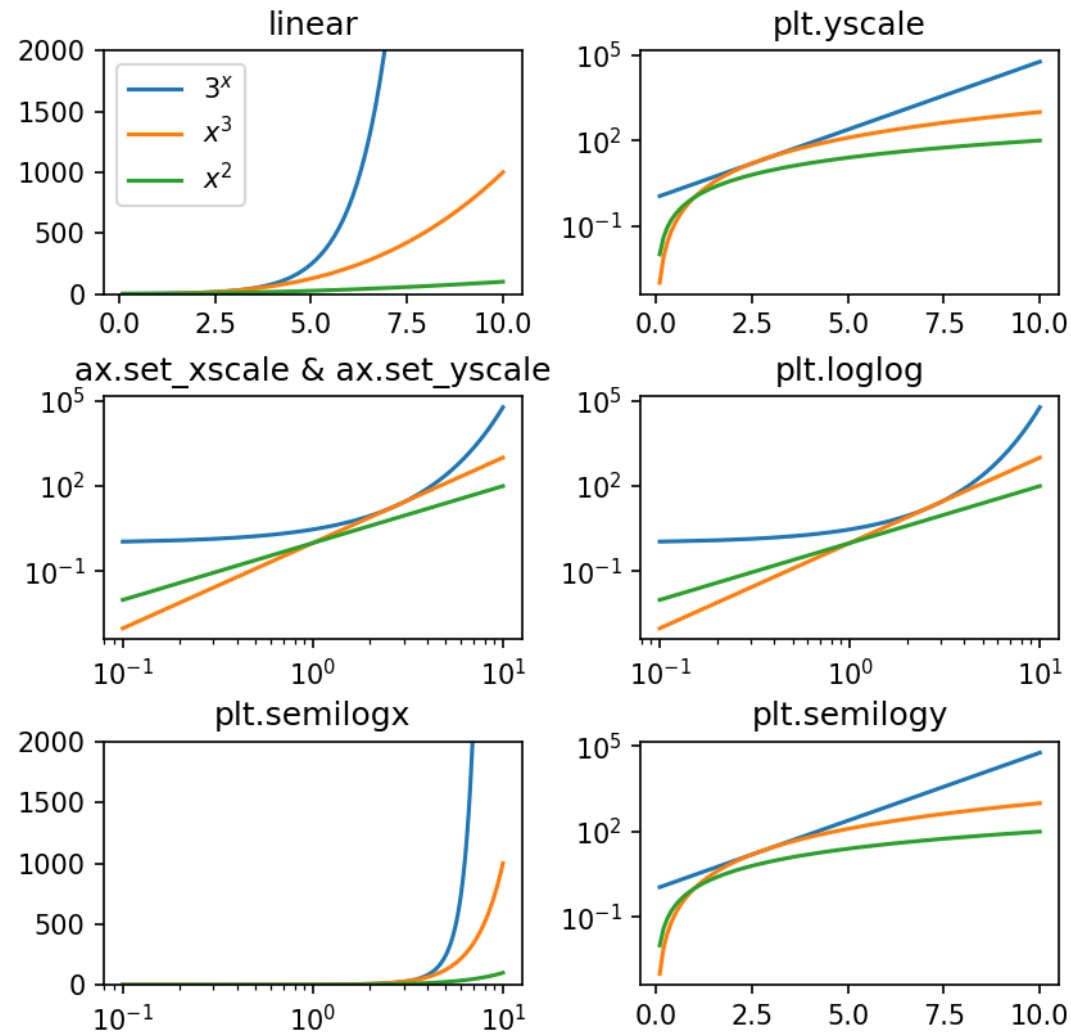
y1 = [i ** 2 for i in x]
y2 = [i ** 3 for i in x]
y3 = [3 ** i for i in x]

for i in range(1, 7):
    ax = plt.subplot(3, 2, i)
    plt.plot(x, y3, label='$3^x$')
    plt.plot(x, y2, label='$x^3$')
    plt.plot(x, y1, label='$x^2$')
    match i:
        case 1:
            plt.ylim(0, 2000)
            plt.xscale('linear') # default
            plt.yscale('linear') # default
            plt.legend()
            plt.title('linear')
        case 2:
            plt.yscale('log')
            plt.title('plt.yscale')
        case 3:
            ax.set_xscale('log')
            ax.set_yscale('log')
            plt.title('ax.set_xscale & ax.set_yscale')
        case 4:
            plt.loglog()
            plt.title('plt.loglog')
        case 5:
            plt.ylim(0, 2000)
            plt.semilogx()
            plt.title('plt.semilogx')
        case 6:
            plt.semilogy()
            plt.title('plt.semilogy')

plt.show()

```

log scales



- There are many ways to make the x- and/or y-axis logarithmic with pyplot

Saving figures

matplotlib-savefig.py

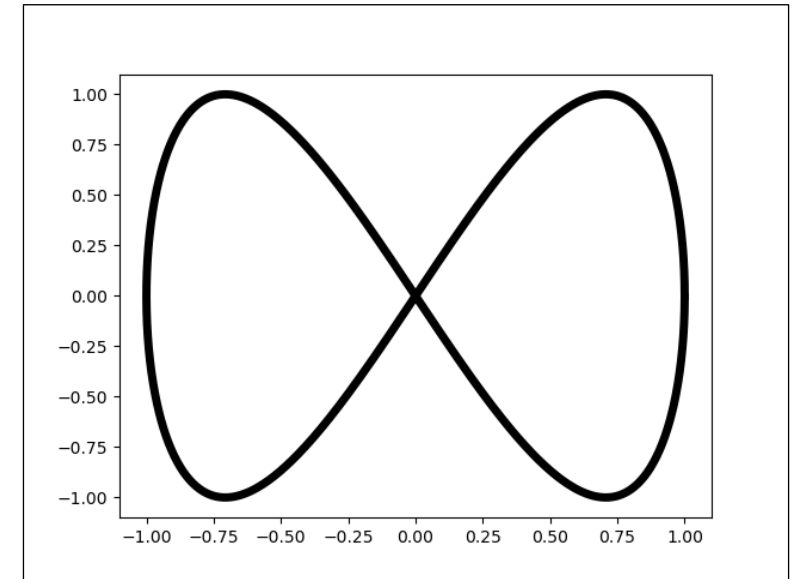
```
import matplotlib.pyplot as plt
from math import pi, sin, cos

n = 1000
points = [(cos(2 * pi * i / n),
           sin(4 * pi * i / n)) for i in range(n)]
x, y = zip(*points)
plt.plot(x, y, 'k-', linewidth=5)

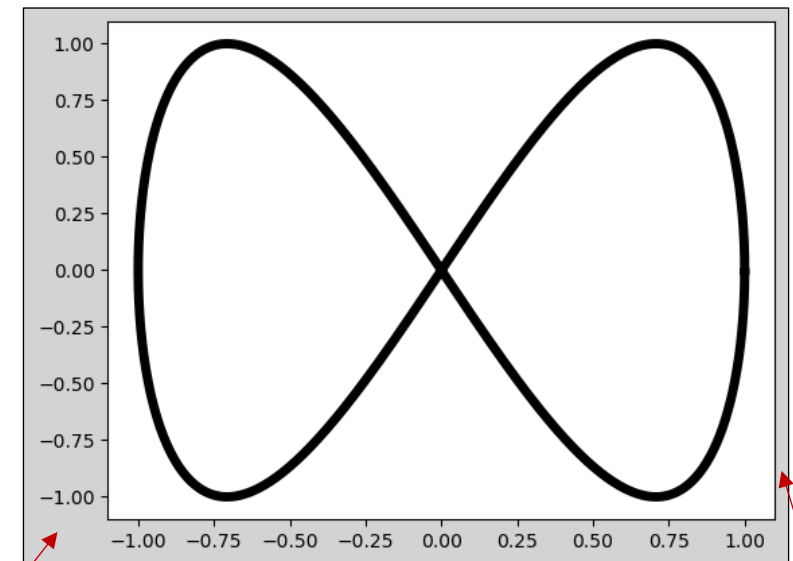
plt.savefig('butterfly.png') # save plot as PNG

plt.savefig('butterfly-grey.png',
            dpi=100,          # dots per inch
            bbox_inches='tight', # crop to bounding box
            pad_inches=0.1,    # space around figure
            facecolor='lightgrey', # background color
            format='png')      # optional if file extension

plt.savefig('butterfly.pdf') # save plot as PDF
plt.show()                  # interactive viewer
```



butterfly.png



butterfly-grey.png

facecolor

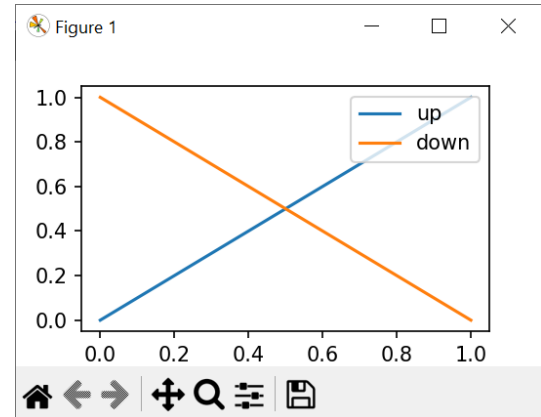
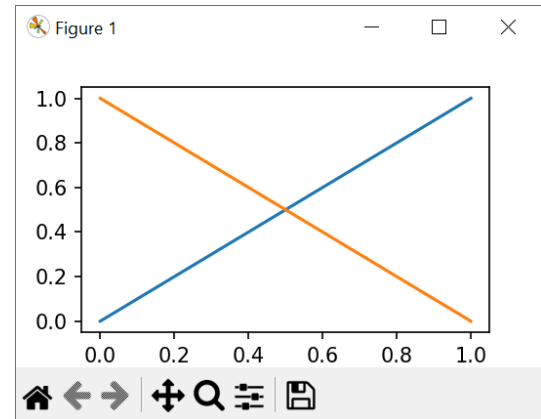
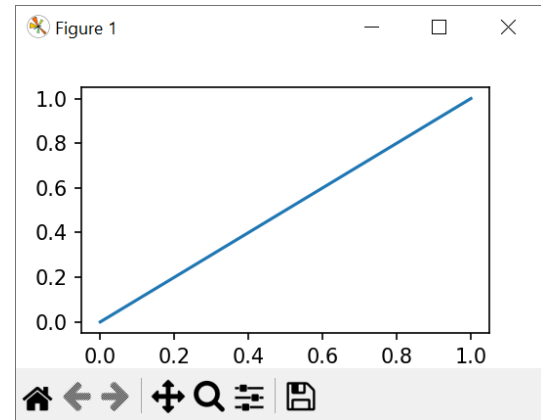
pad_inches

Interactive mode

Python shell

```
> import matplotlib.pyplot as plt
> plt.ion()                                     # Enable interactive mode
> plt.plot([0, 1], [0, 1], label='up')          # Shows plot immediately
> plt.plot([0, 1], [1, 0], label='down')        # Adds visible line
> plt.legend(loc='upper right')                 # Adds visible legend
> plt.ioff()                                    # Disable interactive mode
```

- Useful when developing plot from Python shell
- Automatically shows / updates plot



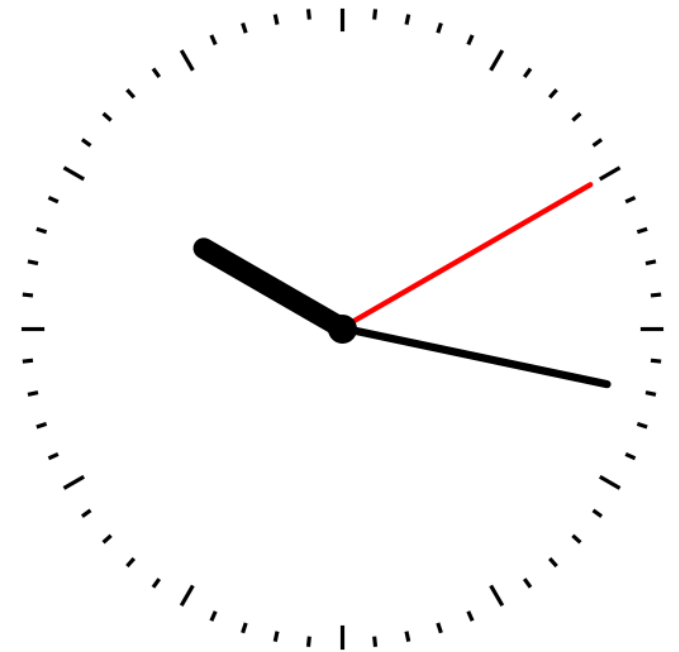
A crude animation

clock.py

```
import matplotlib.pyplot as plt
from math import pi, sin, cos
import datetime

def plot_clock(hour, minute, second):
    plt.axis('off')                # hide x and y axes
    plt.gca().set_aspect('equal')  # don't squeeze circle
    for i in range(60):           # show second marks
        angle = 2 * pi * i / 60
        x, y = cos(angle), sin(angle)
        start = 0.98 if i % 5 else .94 # every 5'th mark should be longer
        plt.plot([start * x, x], [start * y, y], c='black') # mark
    for angle, length, style in [
        (second / 60, 0.9, dict(c='red', lw=2)),
        (minute / 60, 0.85, dict(c='black', lw=3)),
        ((hour + minute / 60) / 12, 0.5, dict(c='black', lw=8))
    ]:
        angle = 2 * pi * (0.25 - angle)
        x, y = length * cos(angle), length * sin(angle)
        plt.plot([0, x], [0, y], **style, solid_capstyle='round') # arm
    plt.plot(0, 0, 'o', ms=10, c='black') # center dot

while True:
    now = datetime.datetime.now() # local time
    plot_clock(now.hour, now.minute, now.second)
    plt.pause(1) # show figure and pause 1 second
    plt.clf()    # clear figure
```



```

import matplotlib.pyplot as plt
from matplotlib.animation import FuncAnimation
from math import pi, cos, sin

n, tail_length = 200, 75
points = [] # tail_length recent points

def point(i):
    t = 2 * pi * i / n
    return (cos(3 * t), sin(2 * t))

fig = plt.figure() # new figure
ax = plt.gca() # get current axes
ax.set_facecolor('black') # set background color
plt.xlim(-1.1, 1.1) # set x-axis range
plt.ylim(-1.1, 1.1) # set y-axis range
plt.xticks([]) # remove x-ticks & labels
plt.yticks([]) # remove y-ticks & labels
plt.title('Moving point') # plot title

x, y = point(0)
plt.plot(x, y, 'w.') # start point
plt.text(x - 0.025, y, 'start', color='w', # text label
         ha='right', va='center') # alignment
tail, = plt.plot([], [], 'w-', alpha=0.5) # init. tail
head, = plt.plot([], [], 'ro') # init. current point

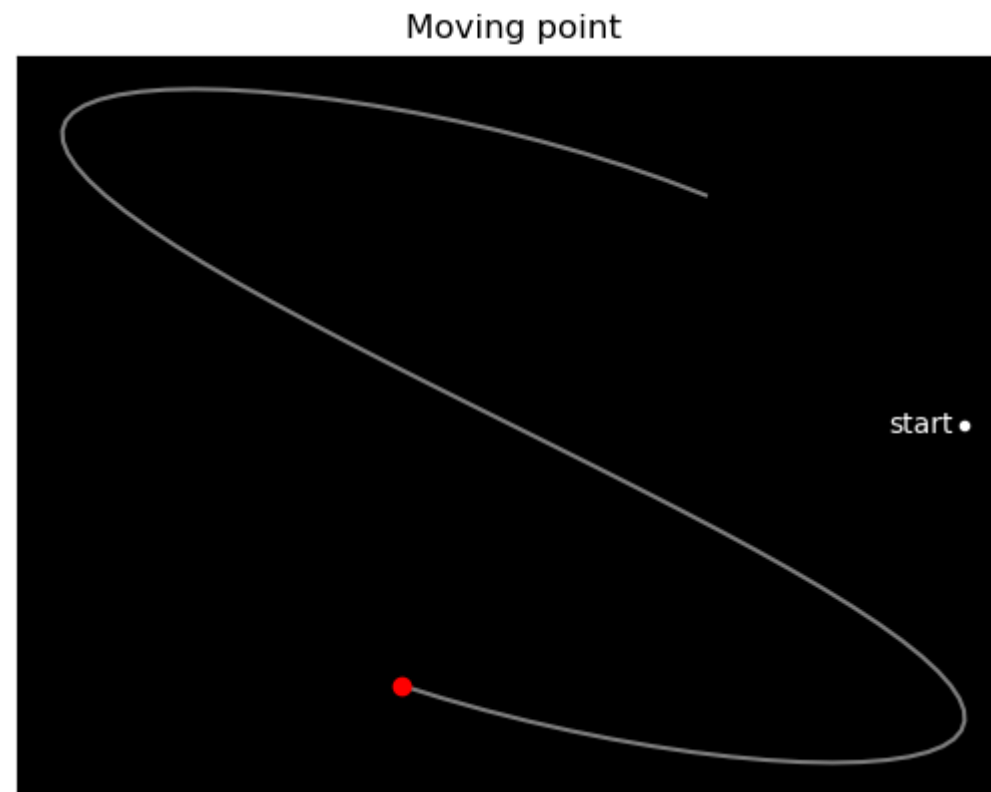
def move(frame): # frame = value from frames
    x, y = point(frame)
    points.append((x, y))
    del points[:-tail_length] # limit tail
    tail.set_data(*zip(*points)) # update tail points
    head.set_data([x], [y]) # update head point

animation = FuncAnimation(fig, # figure to animate
                          func=move, # function called for each frame
                          frames=range(n), # array like to iterate over
                          interval=25, # milliseconds between frames
                          repeat=True, # repeat frames when done
                          repeat_delay=0) # wait milliseconds before repeat

plt.show()

```

matplotlib.animation.FuncAnimation



- `plot` returns “Line2D” objects representing the plotted data
- “Line2D” objects can be updated using `set_data`
- To make an animation you need to repeatedly update the “line2D” objects
- `FuncAnimation` repeatedly calls `func` in regular intervals `interval`, each time with the next value from `frames` (if `frames` is None, then the frame values provided to `func` will be the infinite sequence 0,1,2,3,...)

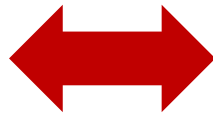


The Jupyter Notebook

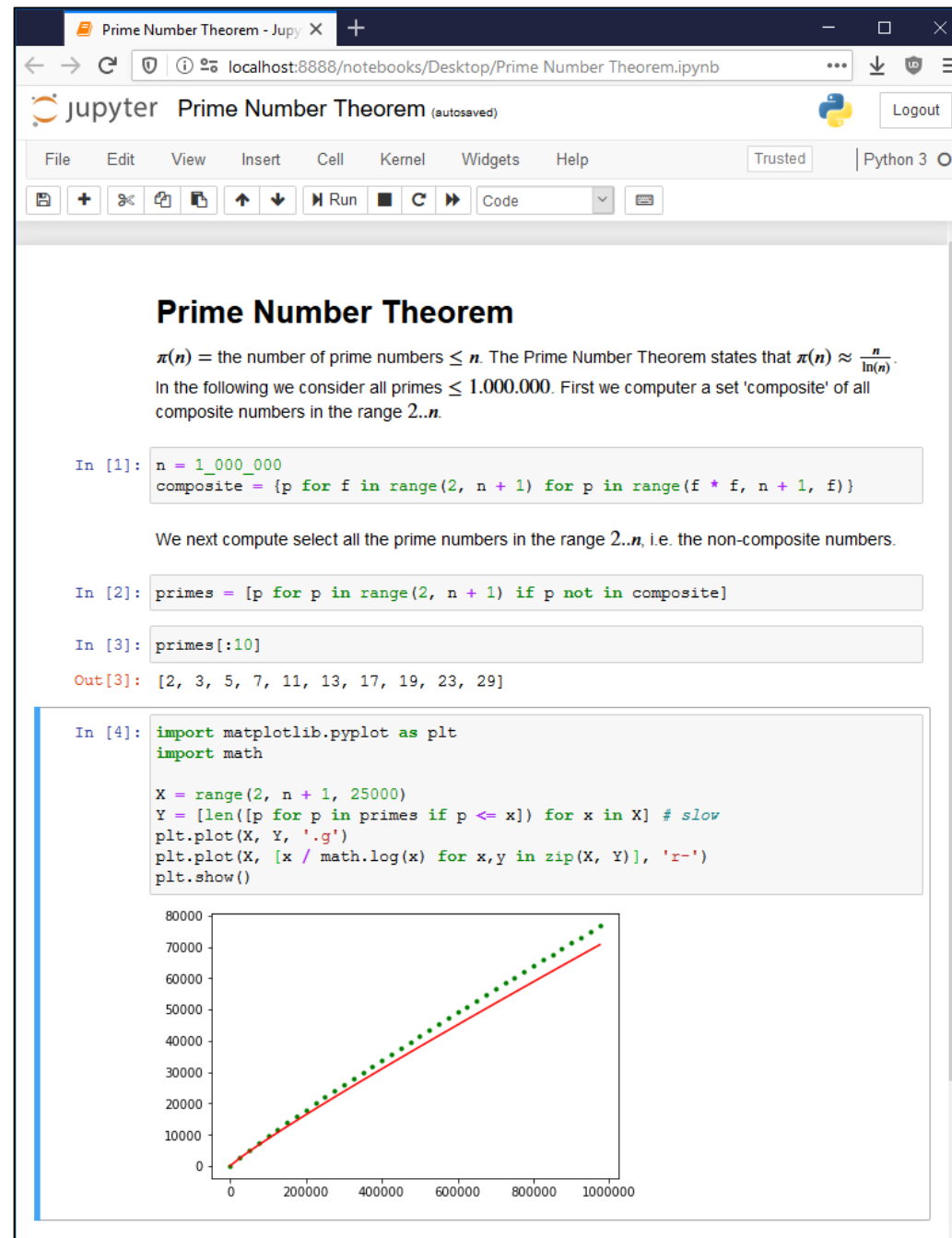
The Jupyter Notebook is an open-source web application that allows you to create and share documents that contain live code, equations, visualizations and narrative text. Uses include: data cleaning and transformation, numerical simulation, statistical modeling, data visualization, machine learning, and much more.



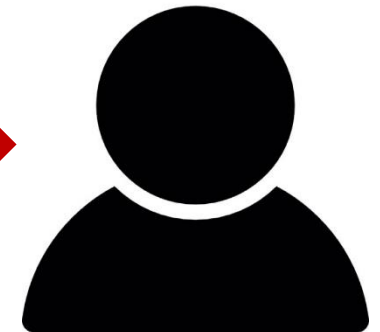
IP[y]:
IPython



Jupyter Server
(e.g. running on
local machine)



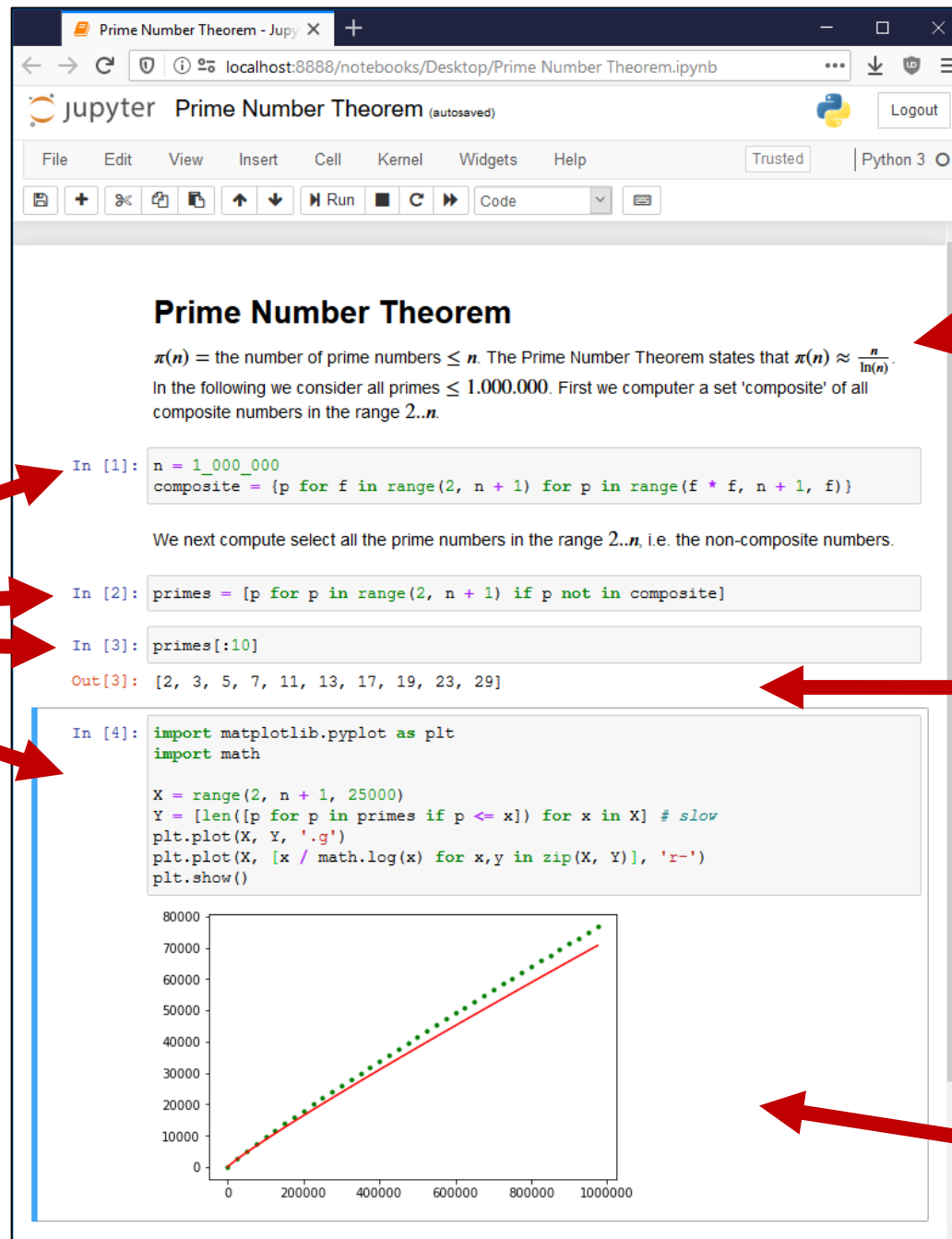
Web Browser



User

cells

python code



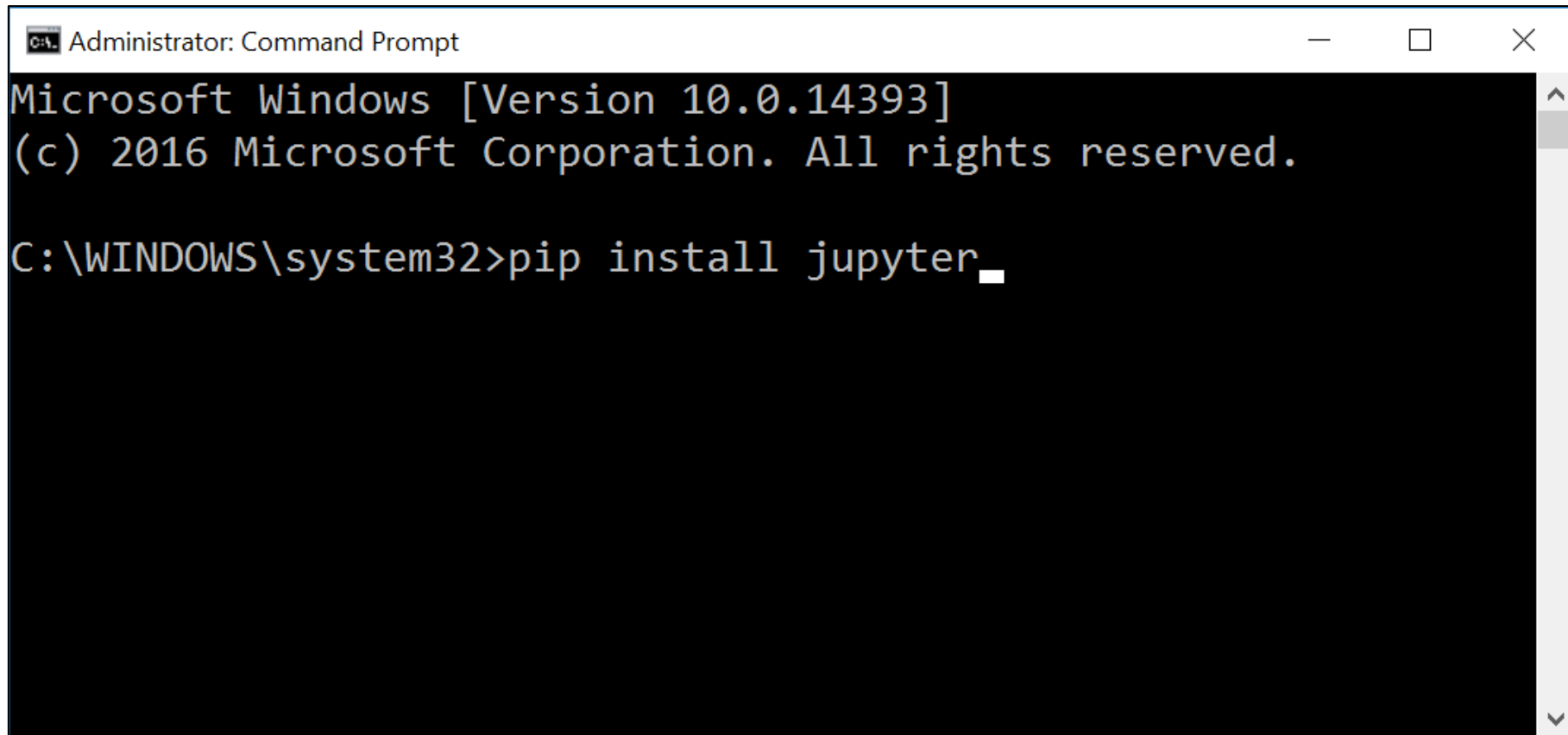
formatted text:
Markdown /
LaTeX / HTML /
...

python shell
output

matplotlib /
numpy / ...
output

Jupyter - installing

- Open a windows shell and run: `pip install jupyter`

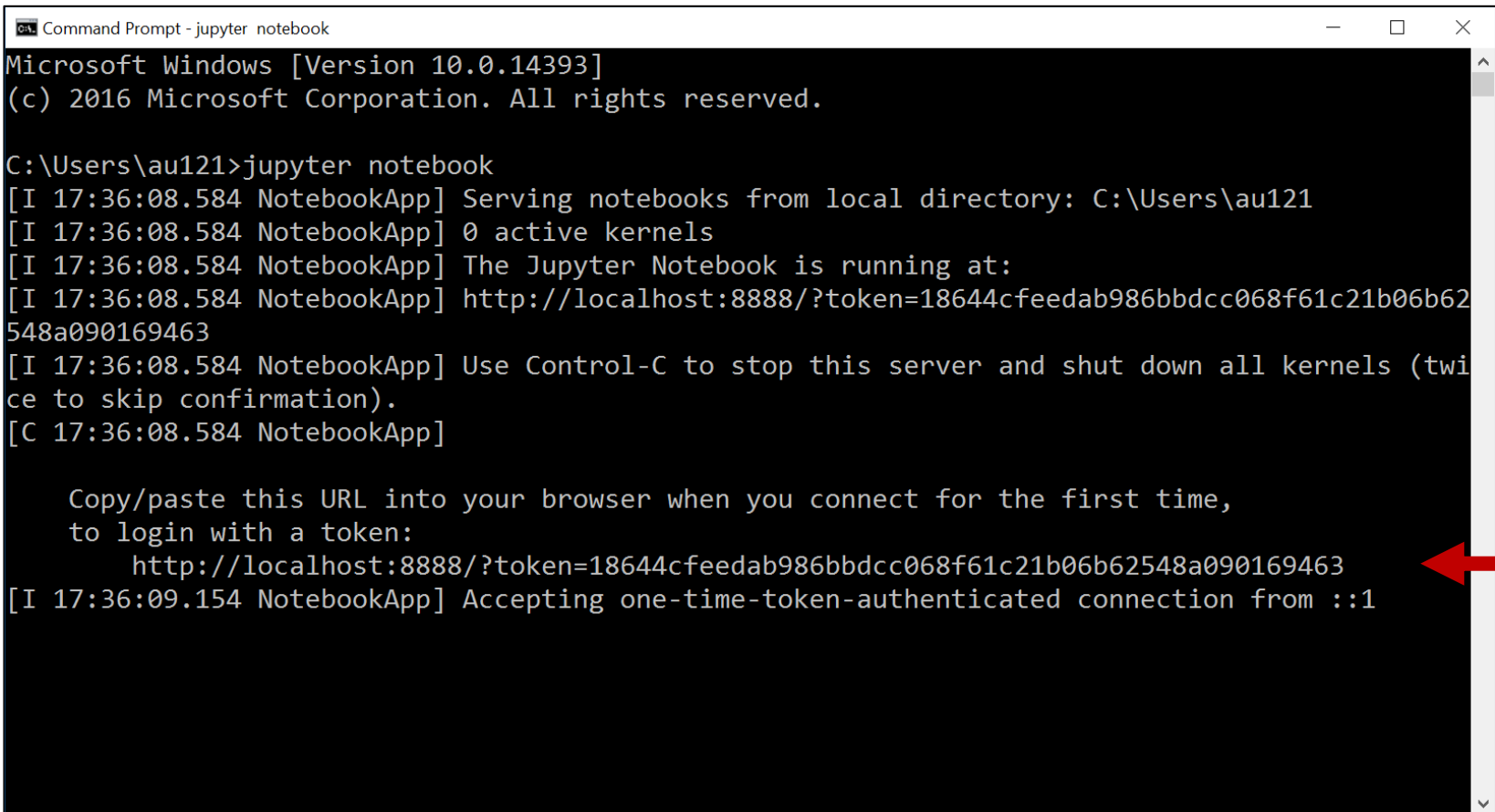


```
Administrator: Command Prompt
Microsoft Windows [Version 10.0.14393]
(c) 2016 Microsoft Corporation. All rights reserved.

C:\WINDOWS\system32>pip install jupyter_
```

Jupyter – launching the jupyter server

- Open a windows shell and run: `jupyter notebook`



```
Command Prompt - jupyter notebook
Microsoft Windows [Version 10.0.14393]
(c) 2016 Microsoft Corporation. All rights reserved.

C:\Users\au121>jupyter notebook
[I 17:36:08.584 NotebookApp] Serving notebooks from local directory: C:\Users\au121
[I 17:36:08.584 NotebookApp] 0 active kernels
[I 17:36:08.584 NotebookApp] The Jupyter Notebook is running at:
[I 17:36:08.584 NotebookApp] http://localhost:8888/?token=18644cfeedab986bbdcc068f61c21b06b62548a090169463
[I 17:36:08.584 NotebookApp] Use Control-C to stop this server and shut down all kernels (twice to skip confirmation).
[C 17:36:08.584 NotebookApp]

Copy/paste this URL into your browser when you connect for the first time,
to login with a token:
    http://localhost:8888/?token=18644cfeedab986bbdcc068f61c21b06b62548a090169463
[I 17:36:09.154 NotebookApp] Accepting one-time-token-authenticated connection from ::1
```

url

- If this does not work, then try `python -m notebook`

Desktop/jupyter/

localhost:8888/tree/Desktop/jupyter

jupyter

Quit Logout

Files Running Clusters

Select items to perform actions on them.

Upload New

0 / Desktop / jupyter

..

Prime Number Theorem.ipynb

Python 3

Text File

Folder

Terminals Unavailable

create new notebook

Untitled - Jupyter Notebook

localhost:8888/notebooks/Desktop/jupyter/Untitled.ipynb?kernel_name=py

jupyter **Untitled** (unsaved changes) Logout

File Edit View Insert Cell Kernel Widgets Help Trusted Python 3

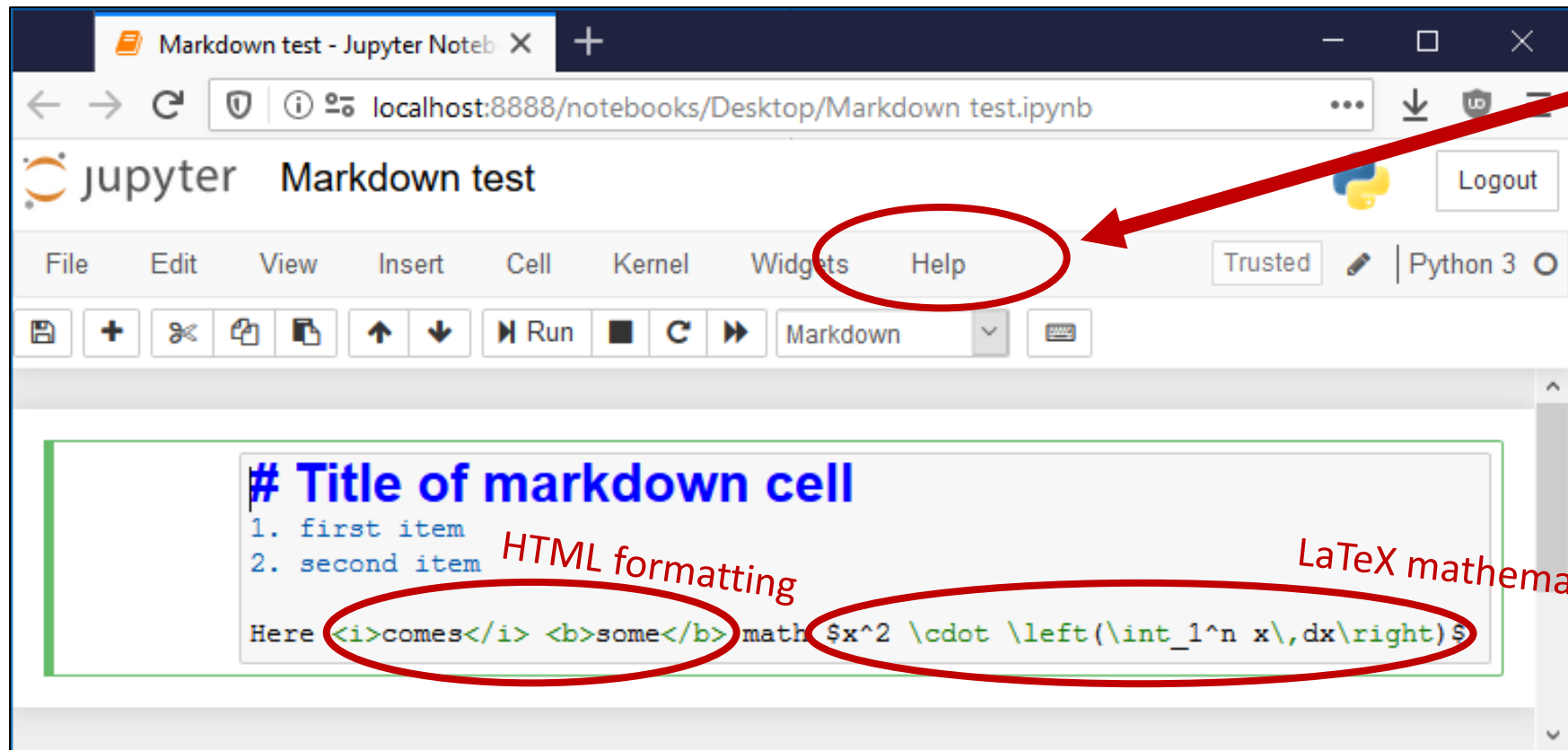
Code

In []: |

title - double click to change

type of active cell

active cell

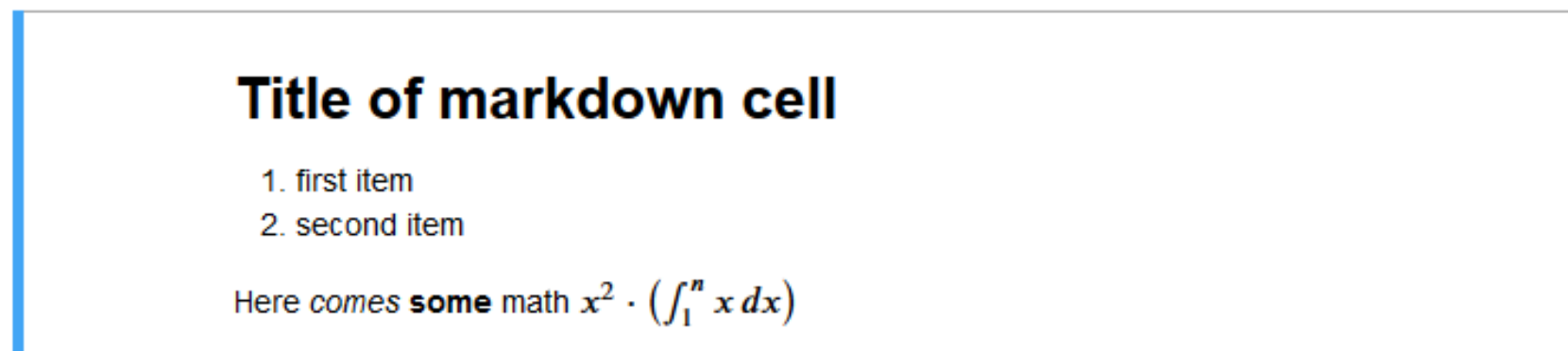


Try:
Help > User Interface Tour
Help > Markdown

HTML formatting

LaTeX mathematics

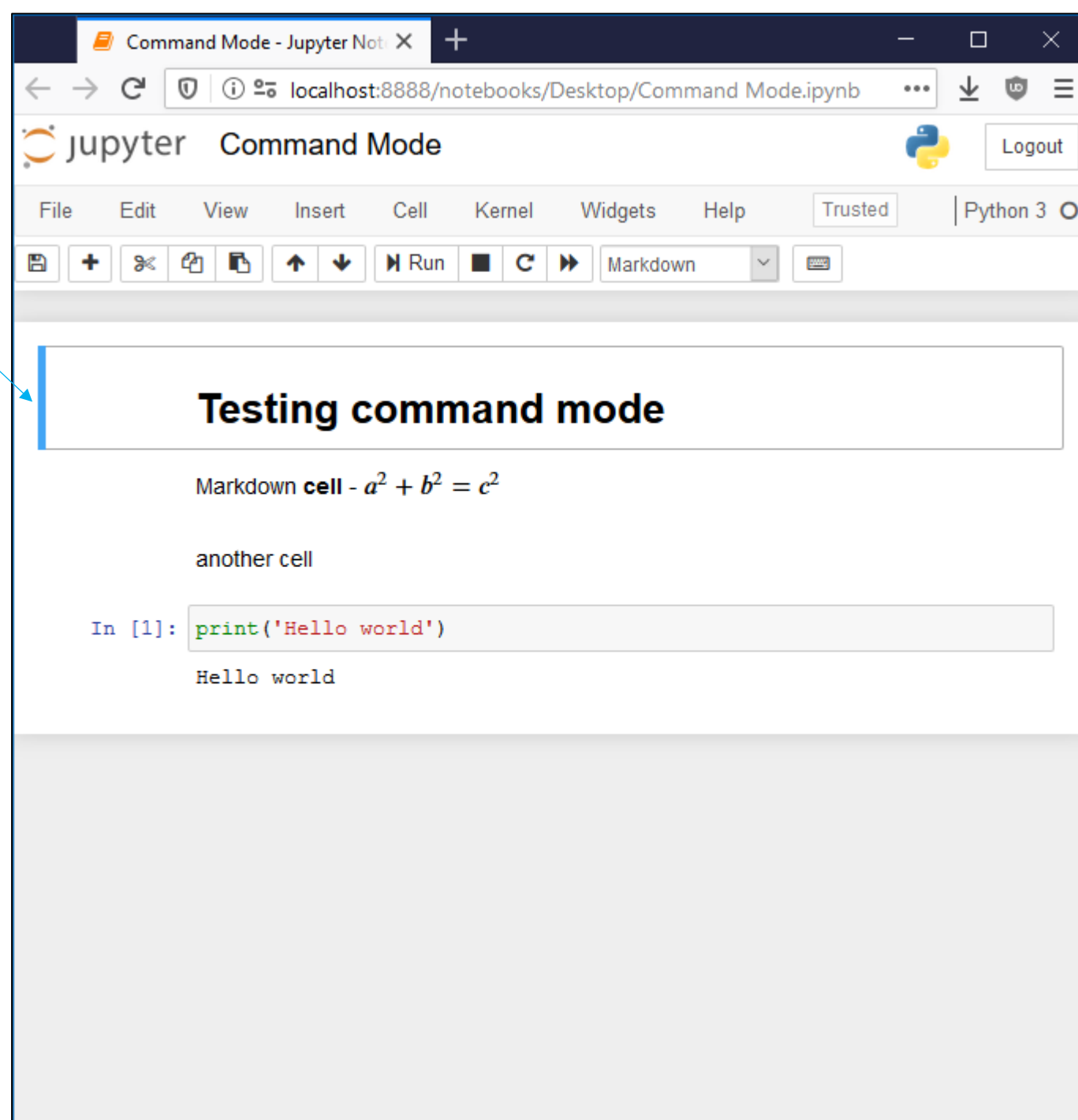
after pressing
Ctrl + Enter (evaluates)
Alt + Enter (evaluates + new cell)



Command Mode

- Used to navigate between cells
- Current cell is marked with blue bar
- Keyboard shortcuts

ctrl-shift-h	show keyboard shortcuts
enter	enter Edit Mode on current cell
escape	leave Edit Mode
shift-enter	run cell + select below
ctrl-enter	run selected cells
alt-enter	run cell and insert below
Y M R	change cell type (code, markdown, raw text)
1 2 3 4 5 6	change heading level
ctrl-A	select all cells
down up	move to next/previous cell
space shift-space	scroll down/up
shift-up shift-down	extend selected cells
A B	insert cell above/below
X C V shift-V Z DD	cut, copy, paste below/above, undo, delete cells
shift-L	toggle line numbers in cells
shift-M	merge selected cells (or with cell below)



The screenshot shows a Jupyter Notebook interface in Command Mode. The browser address bar indicates the notebook is located at `localhost:8888/notebooks/Desktop/Command Mode.ipynb`. The Jupyter logo and the text "Command Mode" are visible in the top left of the interface. A "Logout" button is in the top right. Below the Jupyter logo, there is a menu bar with options: File, Edit, View, Insert, Cell, Kernel, Widgets, and Help. To the right of the menu bar, it says "Trusted" and "Python 3". Below the menu bar is a toolbar with various icons for file operations (save, new, open, save as), navigation (up, down), execution (run, step, interrupt), and cell type selection (code, markdown, raw). The main content area shows a notebook with three cells. The first cell is a markdown cell with the heading "Testing command mode". The second cell is a markdown cell containing the text "Markdown cell - $a^2 + b^2 = c^2$ ". The third cell is a code cell containing the text "another cell" followed by a code input area with `In [1]: print('Hello world')` and its output `Hello world`. A blue vertical bar on the left side of the first cell indicates it is the current cell in Command Mode. A blue arrow points from the text "Current cell is marked with blue bar" in the list to this blue bar.

Edit Mode

- Used to edit current cell
- Current cell is marked with green bar
- Keyboard shortcuts

esc	enter Command Mode
shift-enter	run cell + select below
ctrl-enter	run selected cells
alt-enter	run cell and insert below
ctrl-shift--	split cell at cursor
tab	indent or code completion
shift-tab	show docstring
ctrl-a -x -c -v -z -y	select all, cut, copy, paste, undo, redo
ctrl-d	delete line

The screenshot shows the Jupyter Notebook interface in Edit Mode. The browser address bar indicates the URL is localhost:8888/notebooks/Desktop/Edit Mode.ipynb. The Jupyter logo and 'Edit Mode' text are visible. A menu bar includes File, Edit, View, Insert, Cell, Kernel, Widgets, and Help. A toolbar contains icons for saving, adding, deleting, and running cells, along with a 'Code' dropdown menu. The notebook content includes a title 'Testing edit mode' and a text block 'Here we compute 7 · 6, 2⁸ and 'Hello world''. Below this are three code cells. The first cell contains `print(7 * 6)` and outputs 42. The second cell contains `2 ** 8` and outputs 256. The third cell, which is highlighted with a green border, contains `print('Hello world')`. A tooltip is displayed over this cell, showing the docstring for the `print` function: `Docstring: print(value, ..., sep=' ', end='\n', file=sys.stdout, flush=False)`. A green arrow points from the 'Current cell is marked with green bar' bullet point to the green border of the third cell. A red arrow points from the 'show docstring' shortcut in the table to the docstring tooltip.

Evaluating cells

- To evaluate cell
ctrl-enter, alt-enter, shift-enter
- Output from program shown below cell
- Result of last evaluated line
- Order of code cells evaluated
Note "x ** 2" computed after "x = 4"
- [*] are cells being evaluated / waiting
- [] not yet evaluated
- Recompute all cells top-down
▶▶ or Kernel > Restart & Run all

Evaluation - Jupyter Notebook

localhost:8888/notebooks/Desktop/Evaluation.ipynb

jupyter Evaluation

File Edit View Insert Cell Kernel Widgets Help Trusted Python 3

Run

```
In [1]: print(42)
```

42

```
In [2]: x = 3
```

```
In [5]: x ** 2
```

Out[5]: 16

```
In [4]: x = 4
```

```
In [*]: while True:
        pass
```

```
In [ ]: print('Hello world')
```

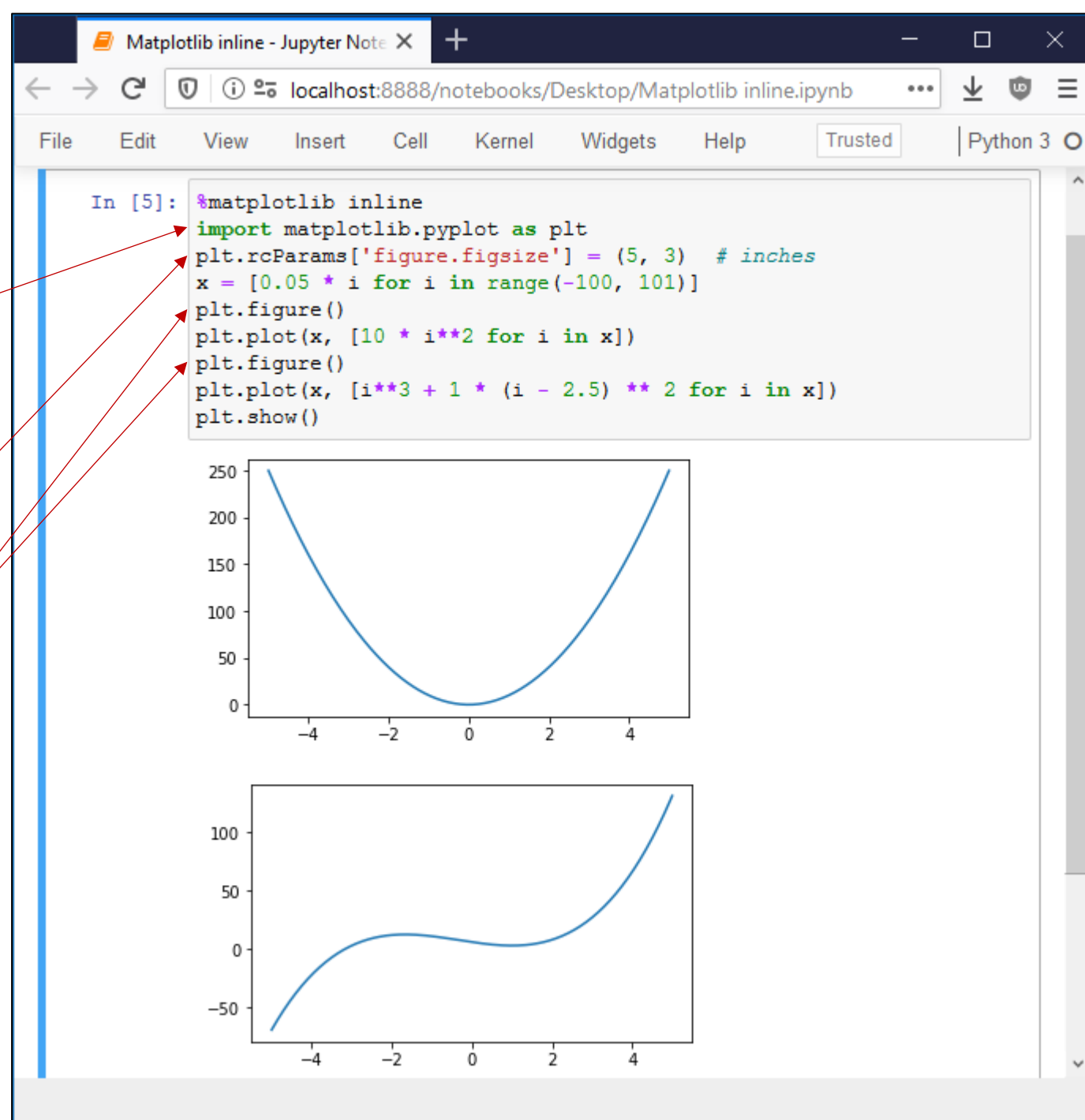
Magic lines

- Jupyter code cells support *magic commands* (actually it is IPython)
- % is a *line magic*
- %% is a *cell magic*

%lsmagic	list magic commands
%quickref	quick reference sheet to IPython
%pwd	print working directory (current folder)
%cd <i>directory</i>	change directory (absolut or relative)
%ls	list content of current directory
%pip or %conda	run pip or conda from jupyter
%load <i>script</i>	insert external script into cell
%run <i>program</i>	run external program and show output
%automagic	toggle if %-prefix is required
%matplotlib inline	no zoom & resize, allows multiple plots
%matplotlib widget	a single plot can be zoomed & resized
%%writefile <i>file</i>	write content of cell to a file
%%time	measure time for cell execution
%timeit <i>expression</i>	time for simple expression

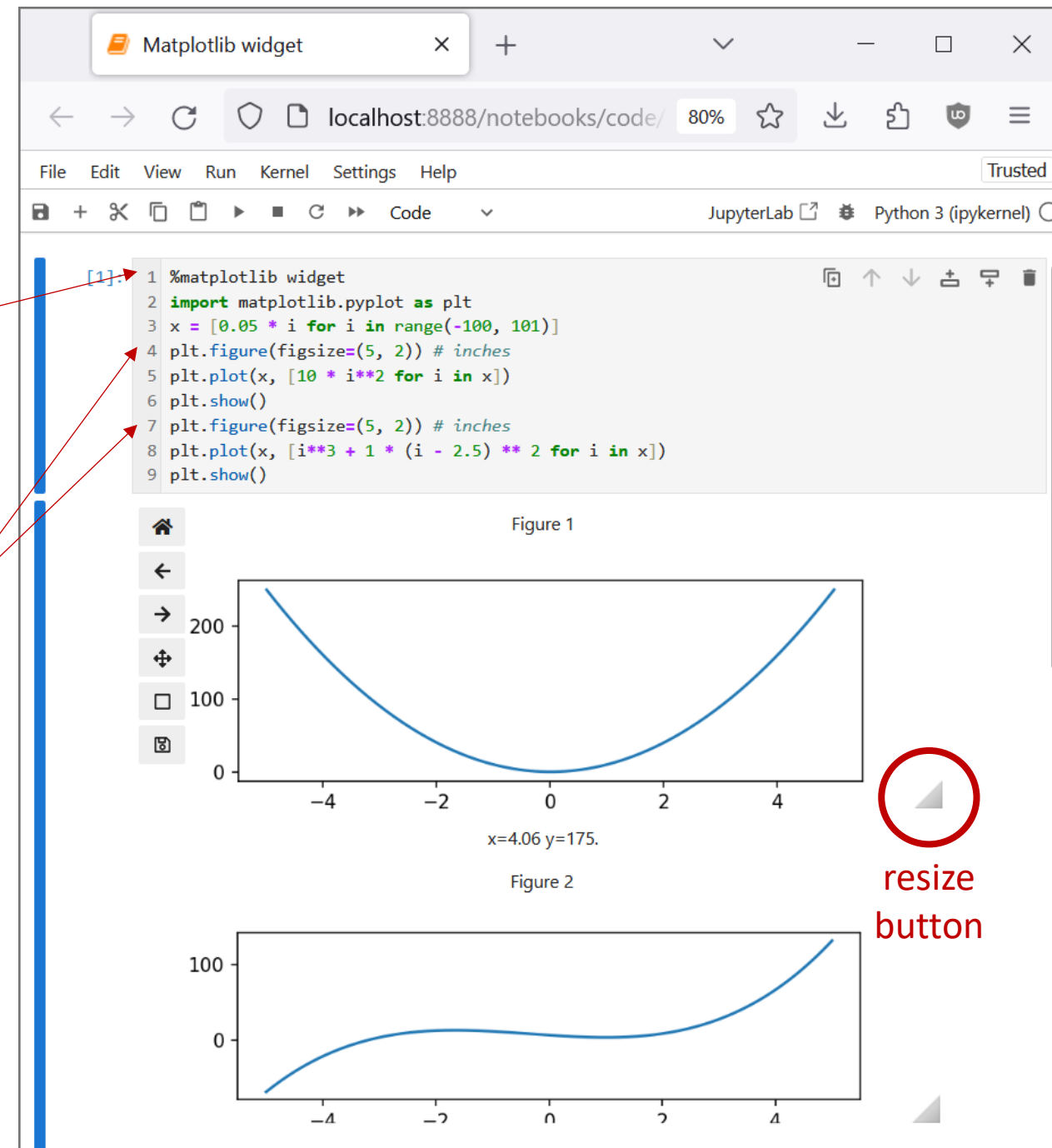
Jupyter and matplotlib

- `%matplotlib inline`
pyplot figures are shown *without* interactive zoom and pan (default)
- Consider changing default figure size
`plt.rcParams['figure.figsize']`
- Start each figure with `plt.figure()`
- Final call to `show` can be omitted



Jupyter and matplotlib

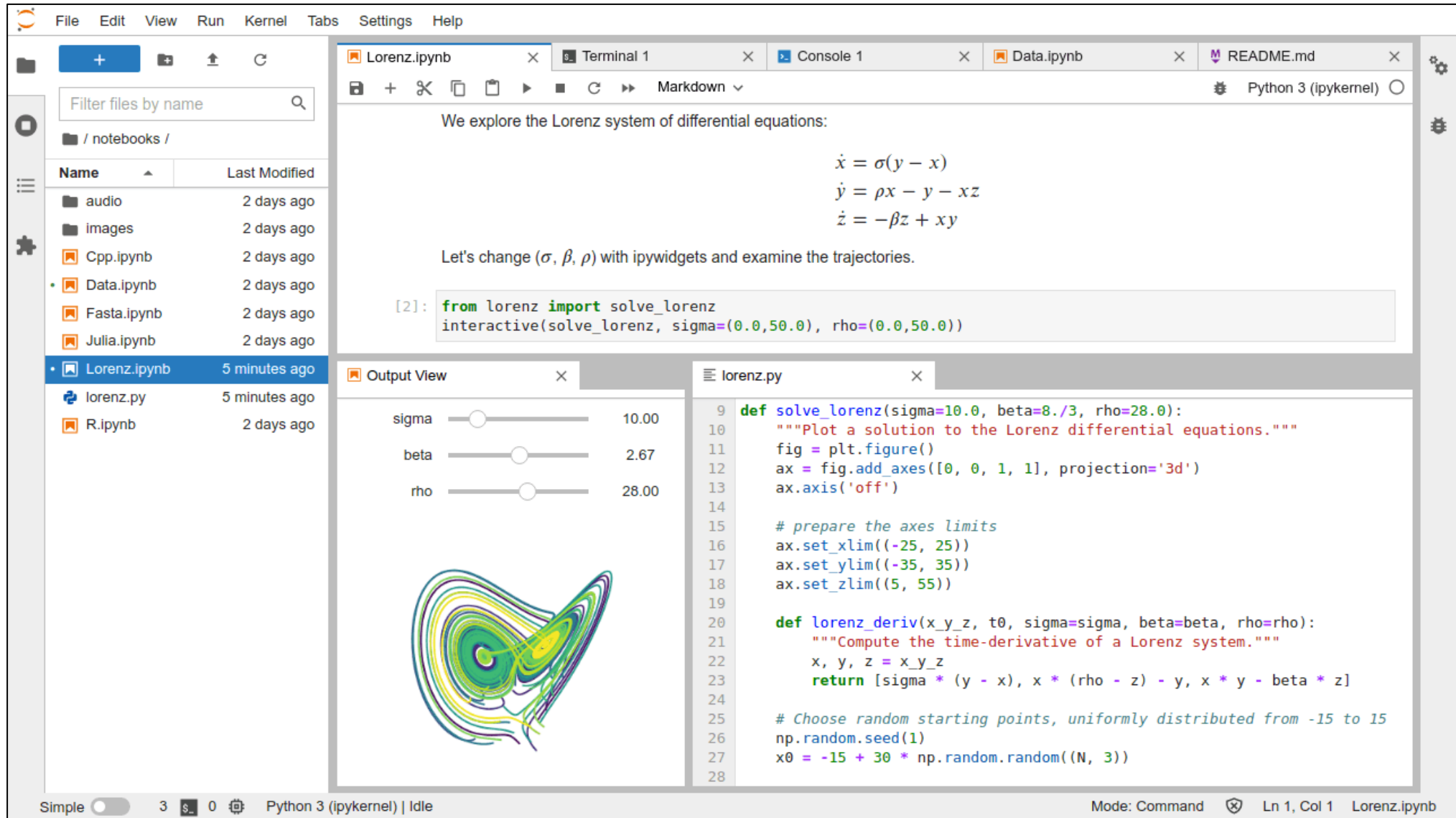
- `%matplotlib widget`
pyplot figures are shown *with* interactive zoom and pan
`pip install ipympl`
- Start each figure with `plt.figure`
(also allows setting figure size)
- Final call to `show` can be omitted





- Widespread tool used for data science applications
- Documentation, code for data analysis, and resulting visualizations are stored in one common format
- Easy to update visualizations
- Works with about 100 different programming languages (not only Python 3), many special features,
- Easy to share data analysis
- IDEs with Notebook support: VS Code, Spyder, PyCharm
- Online Jupyter Notebook with no setup: colab.google
- *Many online tutorials and examples are available*
https://www.youtube.com/results?search_query=jupyter+python

JupyterLab: A Next-Generation Notebook Interface



The screenshot displays the JupyterLab interface with the following components:

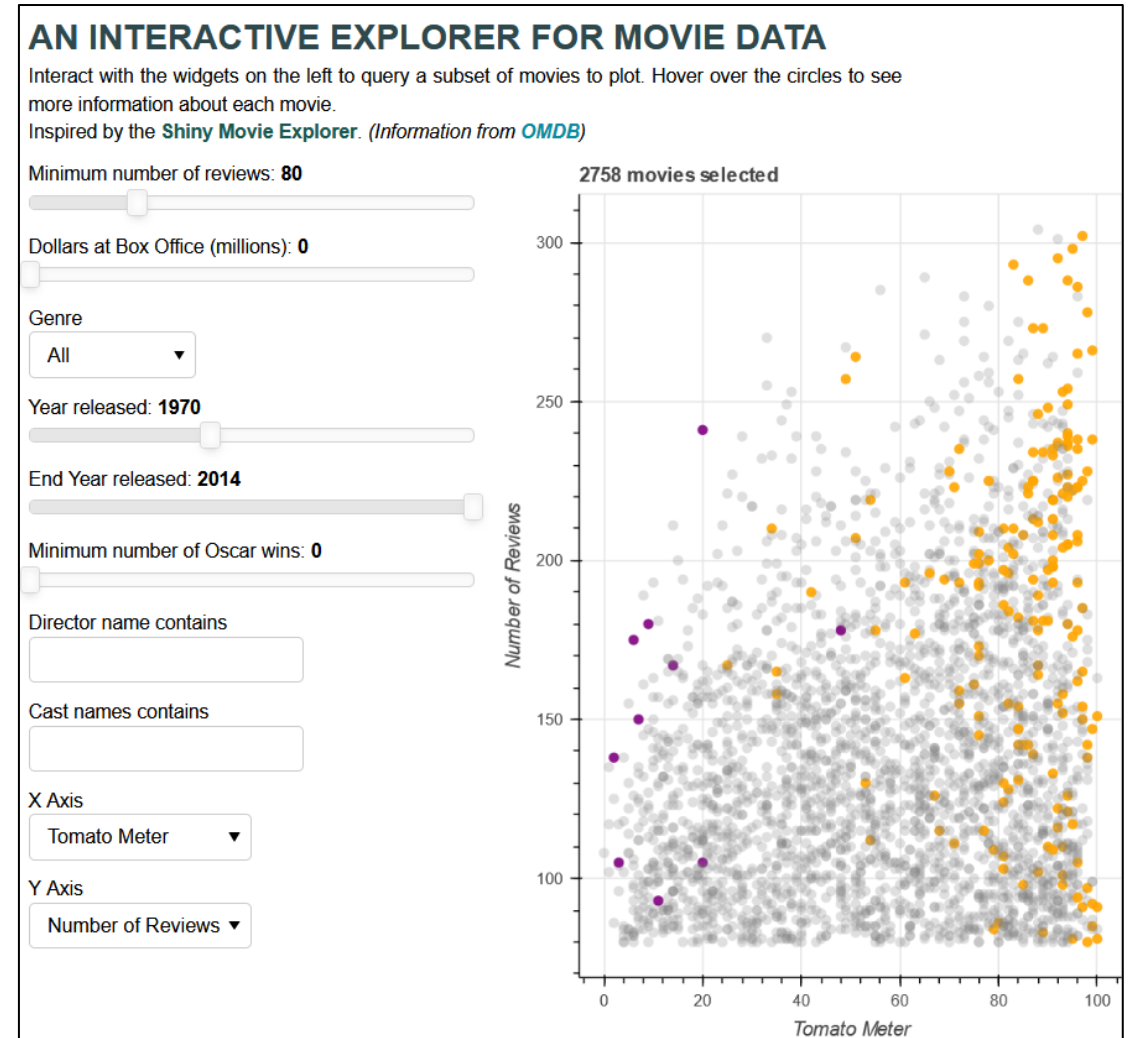
- File Browser (Left):** Shows a file tree with folders like 'audio' and 'images', and notebooks like 'Cpp.ipynb', 'Data.ipynb', 'Fasta.ipynb', 'Julia.ipynb', 'Lorenz.ipynb' (selected), 'lorenz.py', and 'R.ipynb'.
- Notebook (Top):** The 'Lorenz.ipynb' notebook is open, showing a text cell with the Lorenz system equations and a code cell with the following code:

```
[2]: from lorenz import solve_lorenz
      interactive(solve_lorenz, sigma=(0.0,50.0), rho=(0.0,50.0))
```
- Output View (Bottom Left):** Displays interactive sliders for parameters: sigma (0.0 to 10.00), beta (0.0 to 2.67), and rho (0.0 to 28.00). Below the sliders is a 3D plot of the Lorenz attractor.
- Code Editor (Bottom Right):** Shows the source code for 'lorenz.py' with the following code:

```
9 def solve_lorenz(sigma=10.0, beta=8./3, rho=28.0):
10     """Plot a solution to the Lorenz differential equations."""
11     fig = plt.figure()
12     ax = fig.add_axes([0, 0, 1, 1], projection='3d')
13     ax.axis('off')
14
15     # prepare the axes limits
16     ax.set_xlim((-25, 25))
17     ax.set_ylim((-35, 35))
18     ax.set_zlim((5, 55))
19
20     def lorenz_deriv(x_y_z, t0, sigma=sigma, beta=beta, rho=rho):
21         """Compute the time-derivative of a Lorenz system."""
22         x, y, z = x_y_z
23         return [sigma * (y - x), x * (rho - z) - y, x * y - beta * z]
24
25     # Choose random starting points, uniformly distributed from -15 to 15
26     np.random.seed(1)
27     x0 = -15 + 30 * np.random.random((N, 3))
28
```
- Terminal and Console (Top):** The 'Terminal 1' and 'Console 1' tabs are visible but empty.
- Bottom Bar:** Shows the current mode as 'Command', the kernel as 'Python 3 (ipykernel)', and the file name 'Lorenz.ipynb'.



Visualization libraries

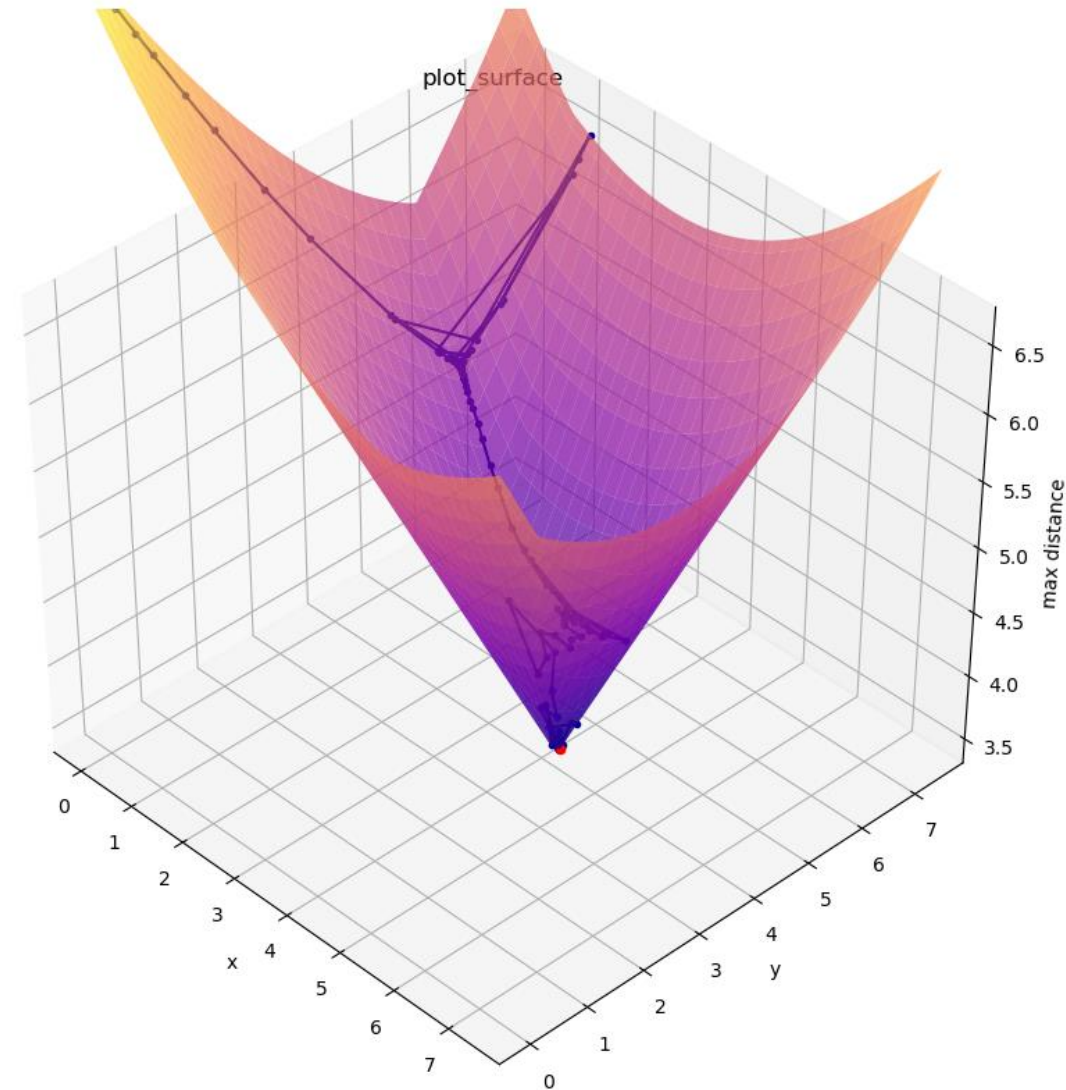
- Altair - altair-viz.github.io
- Bokeh - bokeh.org
- Plotly - plotly.com/python
- Seaborn - seaborn.pydata.org



demo.bokeh.org/movies

scipy.optimize.minimize

- Find point **p** minimizing function **f**
- Supports 13 algorithms – but no guarantee that result is correct
- Knowledge about optimization will help you know what optimization algorithm to select and what parameters to provide for better results
-  **WARNING** 
Many solvers return the wrong value when used as a black box



minimize.py

```
from math import sin
import matplotlib.pyplot as plt
from scipy.optimize import minimize

trace = [] # remember calls to f

def f(x):
    value = x[0]**2 + 10*sin(x[0])
    trace.append((x[0], value))
    return value

X = [-8 + 18 * i / 9999 for i in range(1000)]
Y = [f([x]) for x in X]

plt.style.use('dark_background')
plt.plot(X, Y, 'w-')
for start, color in [(8, 'red'), (-6, 'yellow')]:
    trace = []
    solution = minimize(f, [start], method='nelder-mead')

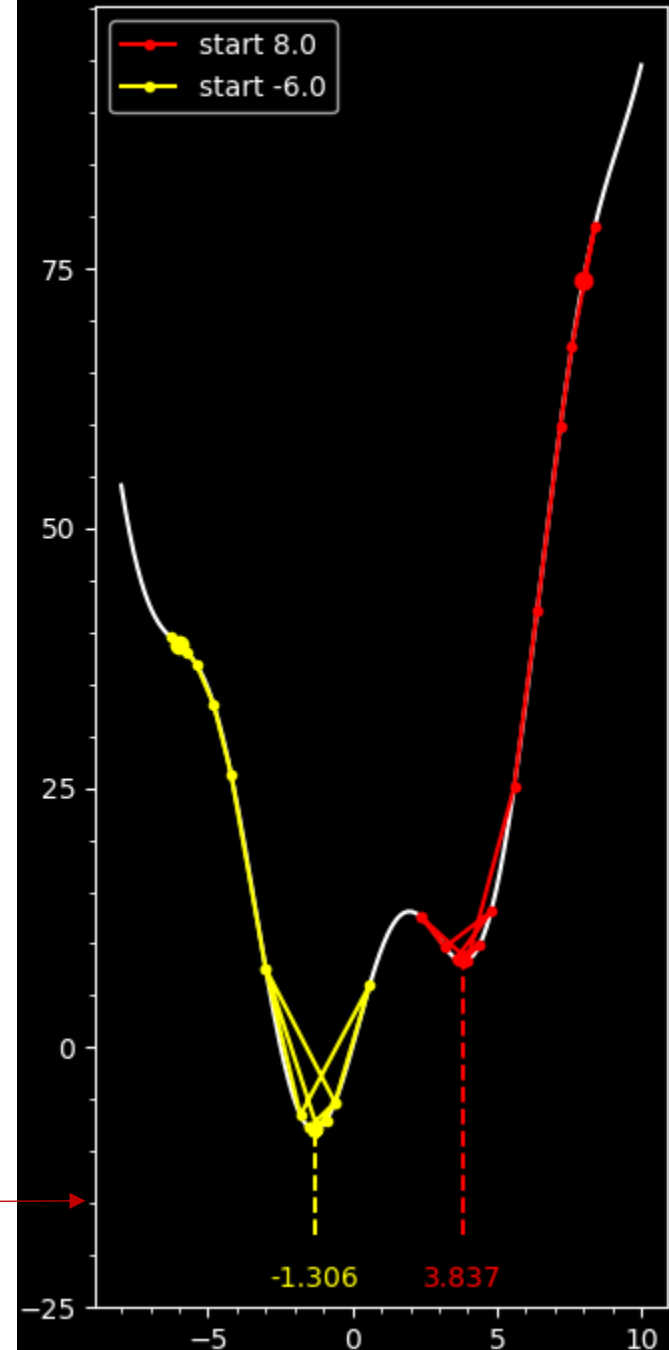
    x, y = solution.x[0], solution.fun
    plt.plot(*zip(*trace), '-.', c=color, label=f'start {start:.1f}') # trace
    plt.plot(*trace[0], 'o', c=color) # first trace point
    plt.text(x, -23, f'{x:.3f}', c=color, ha='center') # show minimum x
    plt.plot([x, x], [-18, y], '--', c=color) # dash to minimum

plt.xticks(range(-5, 15, 5))
plt.yticks(range(-25, 100, 25))
plt.minorticks_on()
plt.legend()
plt.show()
```

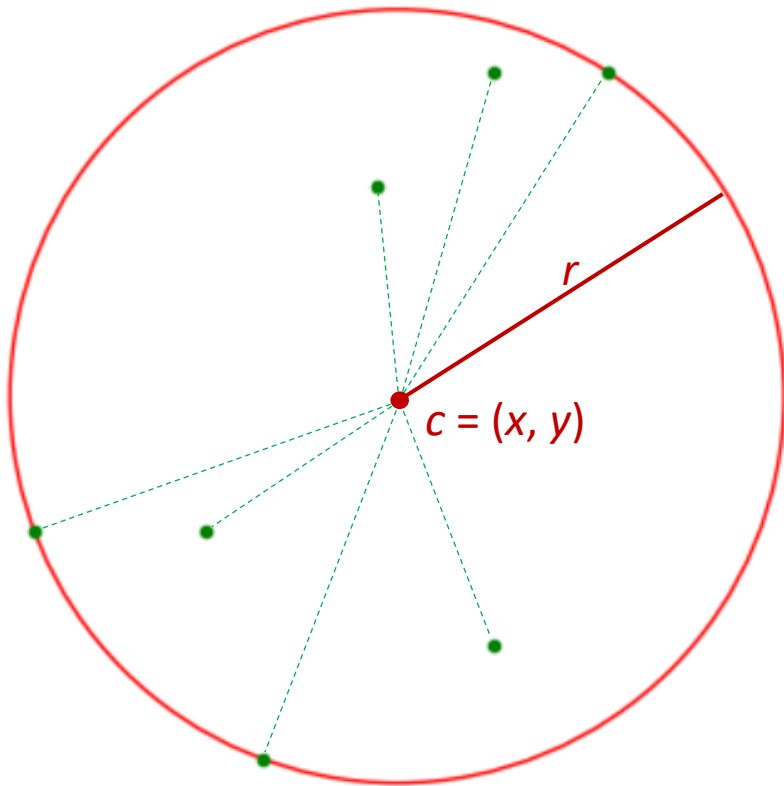
Python shell


```
> print(solution)
final_simplex: (array([[ -1.3064209 ],
 [-1.30649414]]), array([-7.94582337, -7.94582336]))
fun: -7.94582337348758
message: 'Optimization terminated successfully.'
nfev: 38
nit: 19
status: 0
success: True
x: array([-1.3064209])
```

`minimize` tries to find a local minimum for f by repeatedly evaluating f for different x values



Example: Minimum enclosing circle



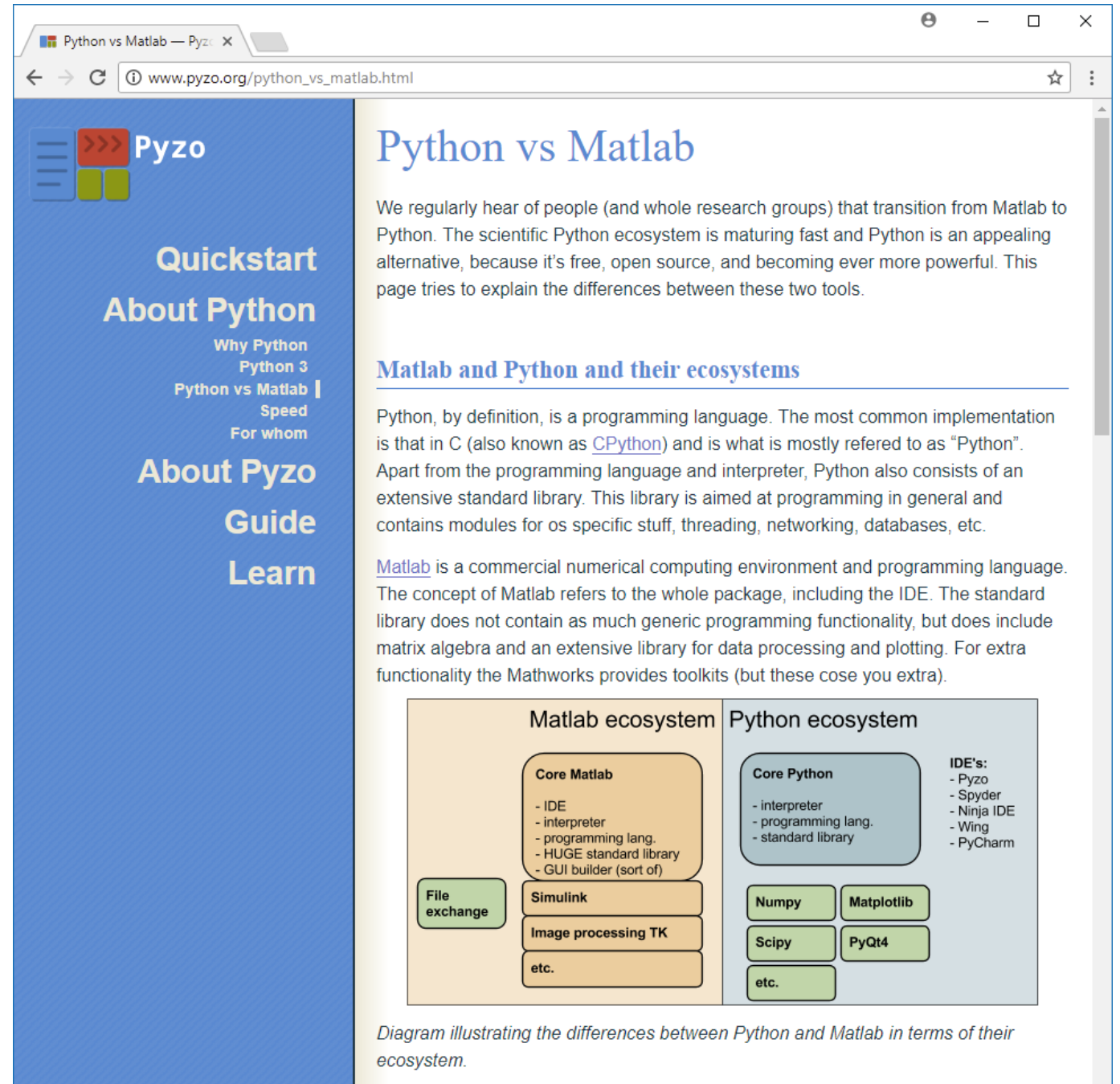
- Find c such that $r = \max_p |p - c|$ is **minimized**
- A solution is characterized by either
 - 1) three points on circle, where the triangle contains the circle center
 - 2) two opposite points on diagonal
- Try a standard numeric minimization solver
-  Computation involves **max** and \sqrt{x} , which can be hard for numeric optimization solvers

Python/scipy vs MATLAB

Some basic differences

- “**end**” closes a MATLAB block
- “**;**” at end of command avoids command output
- $a(i)$ instead $a[i]$
- 1st element of a list $a(1)$
- $a(i:j)$ includes both $a(i)$ and $a(j)$

like R, Mathematica, Julia, AWK, Smalltalk, ...



The screenshot shows a web browser window with the URL www.pyzo.org/python_vs_matlab.html. The page is titled "Python vs Matlab" and features a sidebar with navigation links: "Quickstart", "About Python" (with sub-links "Why Python", "Python 3", "Python vs Matlab", "Speed", "For whom"), "About Pyzo", "Guide", and "Learn".

The main content area is divided into two sections: "Matlab and Python and their ecosystems".

Matlab ecosystem:

- Core Matlab:**
 - IDE
 - interpreter
 - programming lang.
 - HUGE standard library
 - GUI builder (sort of)
- File exchange**
- Simulink**
- Image processing TK**
- etc.**

Python ecosystem:

- Core Python:**
 - interpreter
 - programming lang.
 - standard library
- IDE's:**
 - Pyzo
 - Spyder
 - Ninja IDE
 - Wing
 - PyCharm
- Numpy**
- Matplotlib**
- Scipy**
- PyQt4**
- etc.**

Diagram illustrating the differences between Python and Matlab in terms of their ecosystem.

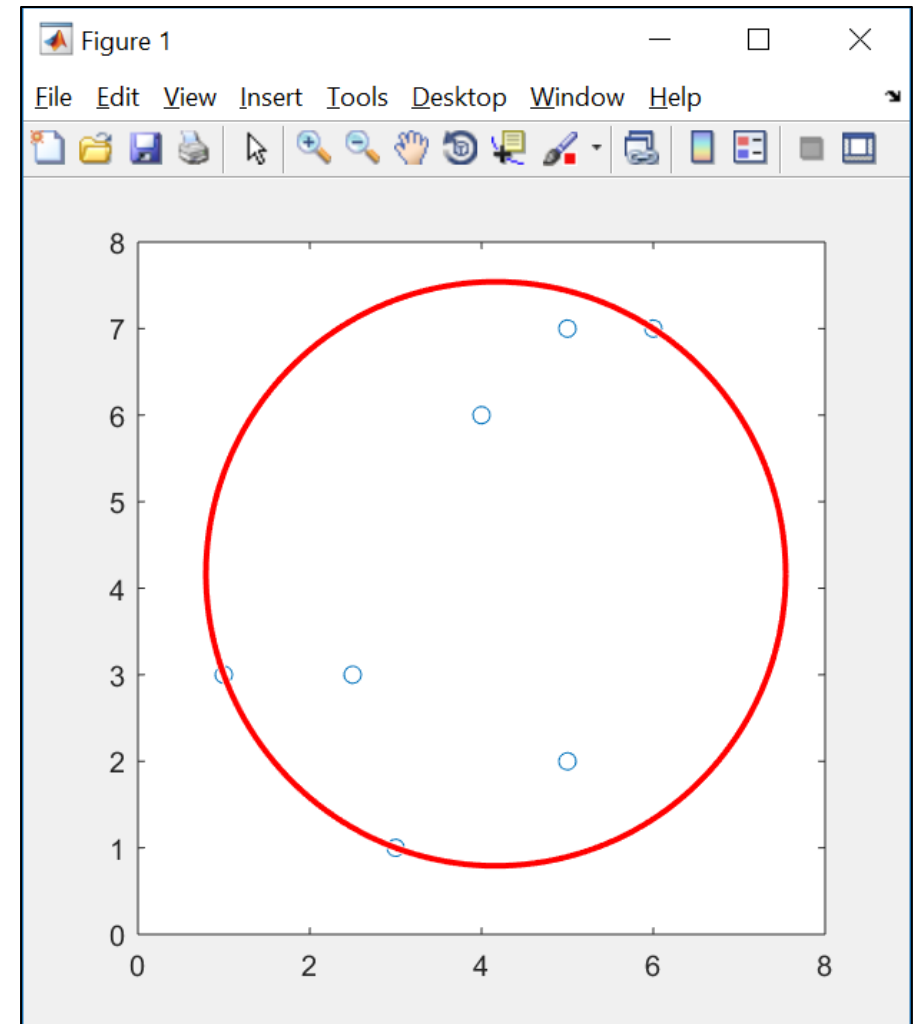
Minimum enclosing circle in MATLAB

enclosing_circle.m

```
% Minimum enclosing circle of a point set
% fminsearch uses the Nelder-Mead algorithm

global x y
x = [1.0, 3.0, 2.5, 4.0, 5.0, 6.0, 5.0];
y = [3.0, 1.0, 3.0, 6.0, 7.0, 7.0, 2.0];
c = fminsearch(@(x) max_distance(x), [0,0]);
plot(x, y, "o");
viscircles(c, max_distance(c));

function dist = max_distance(p)
    global x y
    dist = 0.0;
    for i=1:length(x)
        dist = max(dist, pdist([p; x(i), y(i)],
                                'euclidean'));
    end
end
end
```



Minimum enclosing circle in MATLAB (trace)

enclosing_circle_trace.m

```
global x y trace_x trace_y

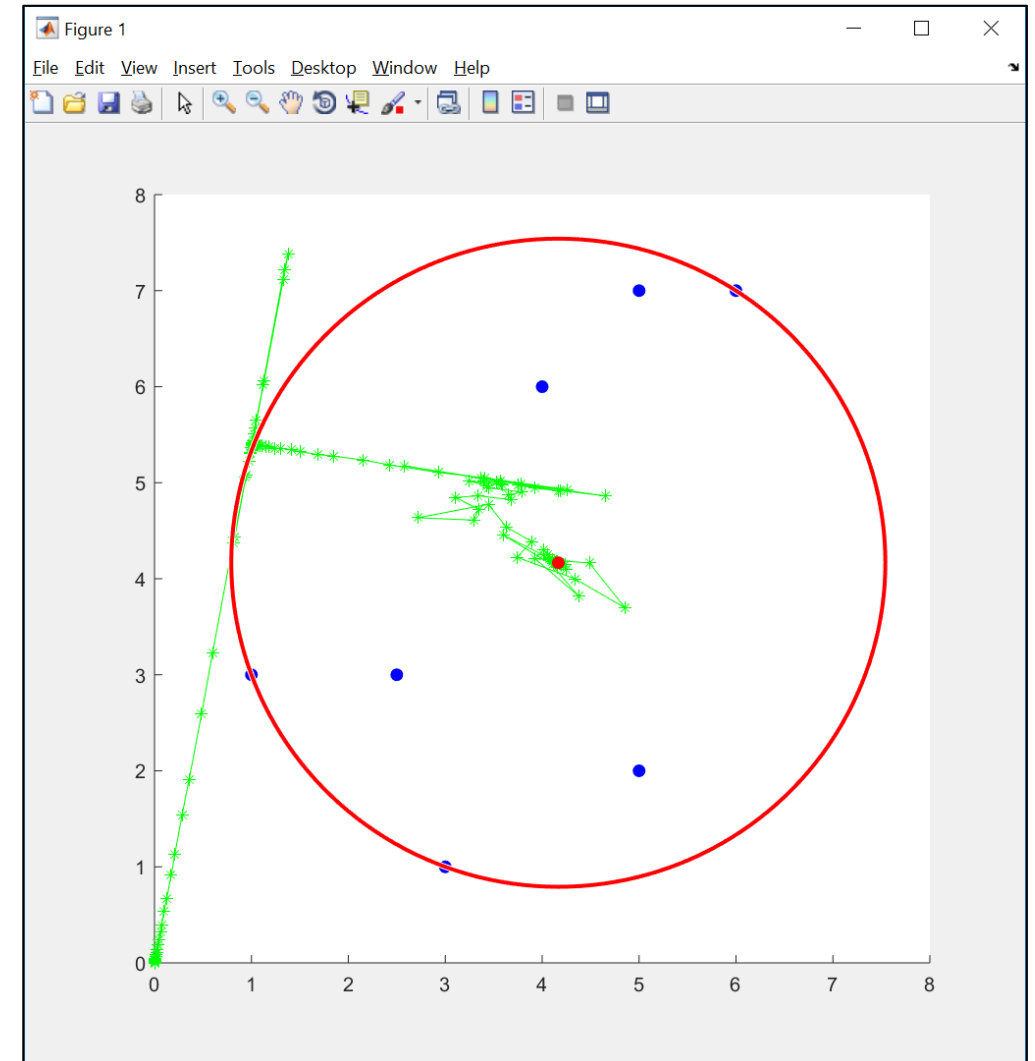
x = [1.0, 3.0, 2.5, 4.0, 5.0, 6.0, 5.0];
y = [3.0, 1.0, 3.0, 6.0, 7.0, 7.0, 2.0];
trace_x = [];
trace_y = [];

c = fminsearch(@(x) max_distance(x), [0, 0]);

hold on
plot(x, y, "o", 'color', 'b', 'MarkerFaceColor', 'b');
plot(trace_x, trace_y, "*-", "color", "g");
plot(c(1), c(2), "o", 'color', 'r', 'MarkerFaceColor', 'r');
viscircles(c, max_distance(c), "color", "red");

function dist = max_distance(p)
    global x y trace_x trace_y
    trace_x = [trace_x, p(1)];
    trace_y = [trace_y, p(2)];

    dist = 0.0;
    for i=1:length(x)
        dist = max(dist, pdist([p; x(i), y(i)], 'euclidean' ));
    end
end
```



Minimum enclosing circle in Python

enclosing_circle.py

```
from scipy.optimize import minimize } import modules
import matplotlib.pyplot as plt

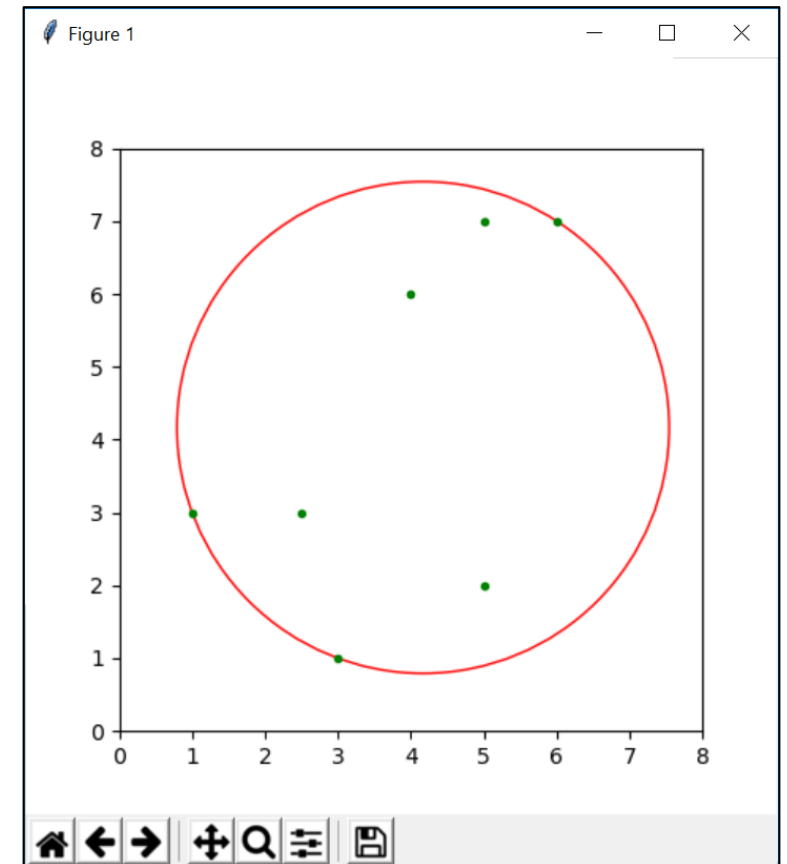
x = [1.0, 3.0, 2.5, 4.0, 5.0, 6.0, 5.0]
y = [3.0, 1.0, 3.0, 6.0, 7.0, 7.0, 2.0]

def dist(p, q):
    return ((p[0] - q[0]) ** 2 + (p[1] - q[1]) ** 2) ** 0.5
def max_distance(c):
    return max([dist(p, c) for p in zip(x, y)])

c = minimize(max_distance, [0.0, 0.0], method='nelder-mead').x
ax = plt.gca()
ax.set_xlim((0, 8))
ax.set_ylim((0, 8))
ax.set_aspect('equal')
plt.plot(x, y, 'g.')
ax.add_artist(plt.Circle(c, max_distance(c),
                        color='r', fill=False))
plt.show()
```

optimization method

manually set axis (force circle inside plot)



Minimum enclosing circle in Python (trace)

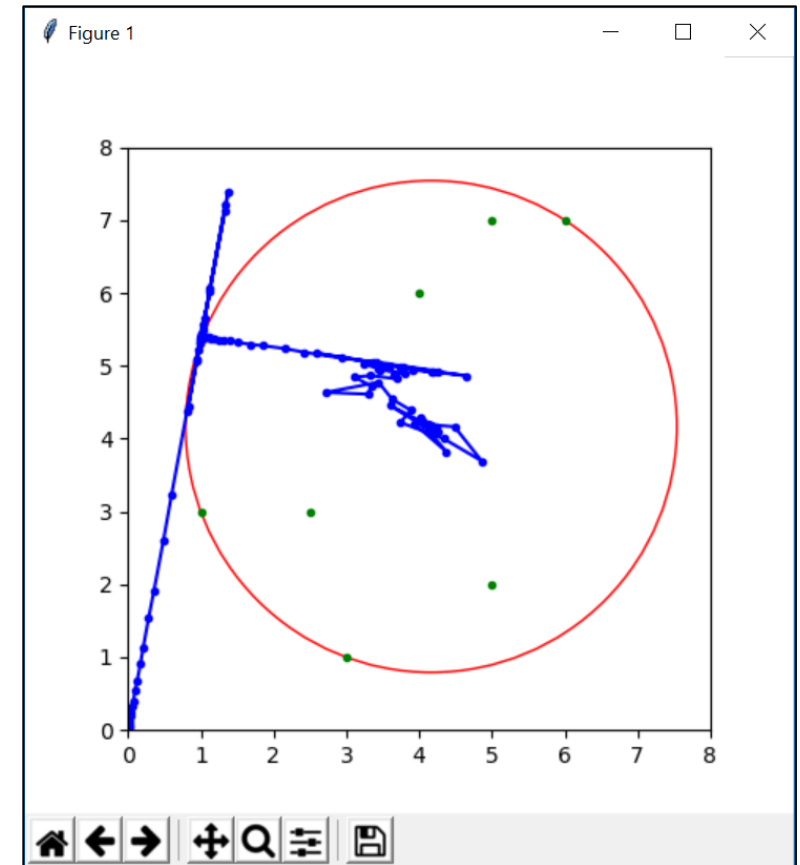
enclosing_circle_trace.py

```
from scipy.optimize import minimize
import matplotlib.pyplot as plt

x = [1.0, 3.0, 2.5, 4.0, 5.0, 6.0, 5.0]
y = [3.0, 1.0, 3.0, 6.0, 7.0, 7.0, 2.0]
trace = []

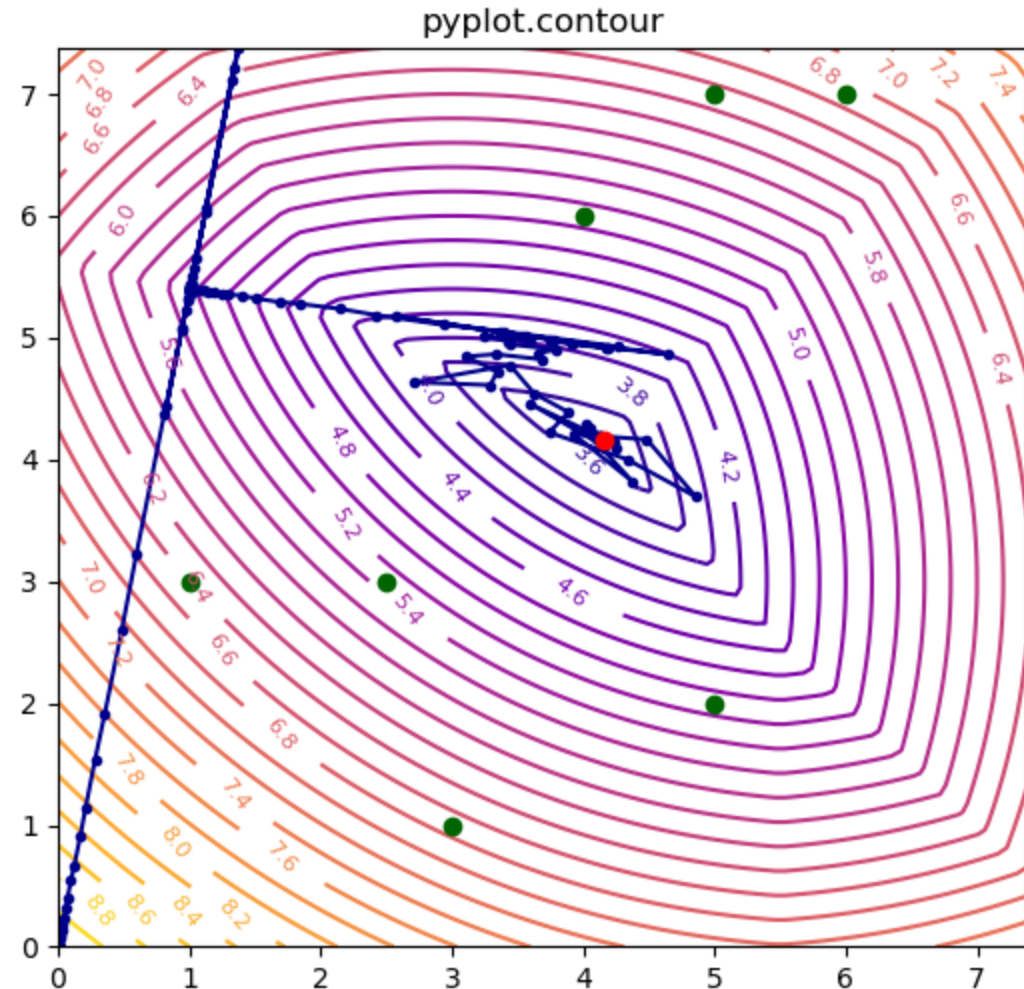
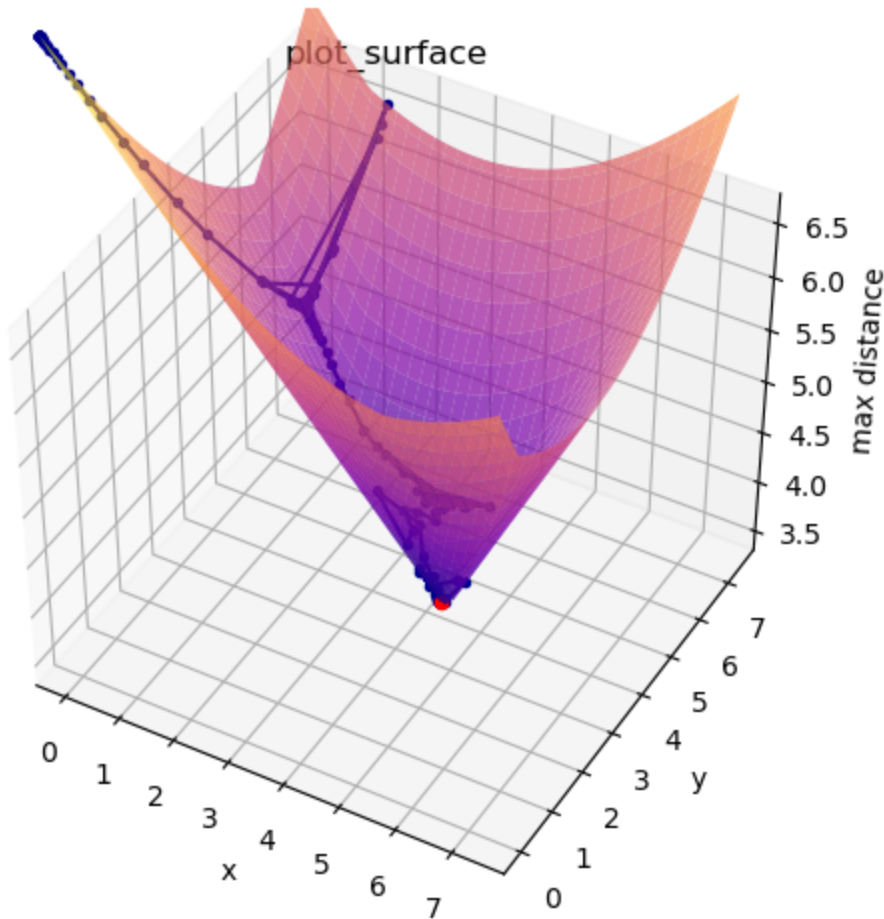
def dist(p, q):
    return ((p[0] - q[0]) ** 2 + (p[1] - q[1]) ** 2) ** 0.5
def max_distance(c):
    trace.append(c)
    return max([dist(p, c) for p in zip(x, y)])

c = minimize(max_distance, [0.0, 0.0],
             method='nelder-mead').x
ax = plt.gca()
ax.set_xlim((0, 8))
ax.set_ylim((0, 8))
ax.set_aspect("equal")
plt.plot(x, y, "g.")
plt.plot(*zip(*trace), 'b.-')
ax.add_artist(plt.Circle(c, max_distance(c),
                        color='r', fill=False))
plt.show()
```



Minimum enclosing circle – search space

Maximum distance to an input point



enclosing_circle_search_space.py (previous slide)

```
from scipy.optimize import minimize
import matplotlib.pyplot as plt
import numpy as np
from mpl_toolkits.mplot3d import Axes3D

points = [(1.0, 3.0), (3.0, 1.0), (2.5, 3.0),
          (4.0, 6.0), (5.0, 7.0), (6.0, 7.0), (5.0, 2.0)]

# Minimum enclosing circle solver
trace = []

def distance(p, q):
    return ((p[0]-q[0])**2 + (p[1]-q[1])**2)**0.5

def distance_max(q):
    dist = max([distance(p, q) for p in points])
    trace.append((q, dist))
    return dist

solution = minimize(distance_max, [0.0, 0.0],
                    method='nelder-mead')

center = solution.x
radius = solution.fun

# unzip point coordinates
points_x, points_y = zip(*points)
trace_x, trace_y, trace_z = zip(*trace)

# Bounding box [x_min, x_max] x [y_min, y_max]
xs, ys = points_x + trace_x, points_y + trace_y
x_min, x_max = min(xs), max(xs)
y_min, y_max = min(ys), max(ys)
# enforce aspect ratio
x_max = max(x_max, x_min + y_max - y_min)
y_max = max(y_max, y_min + x_max - x_min)
```

```
# Minimum enclosing circle - 3D surface plot
# (plot_surface requires X, Y, Z are 2D numpy.arrays)
X, Y = np.meshgrid(np.linspace(x_min, x_max, 100),
                   np.linspace(y_min, y_max, 100))
Z = np.zeros(X.shape)
for px, py in points:
    Z = np.maximum(Z, (X - px)**2 + (Y - py)**2)
Z = np.sqrt(Z)

ax = plt.subplot(1, 2, 1, projection='3d')
ax.plot_surface(X, Y, Z, cmap='plasma', alpha=0.7)
ax.plot(trace_x, trace_y, trace_z, '.-', c='darkblue')
ax.scatter(*center, radius, 'o', c='red')
ax.set_xlabel('x')
ax.set_ylabel('y')
ax.set_zlabel('max distance')
ax.set_title('plot_surface')

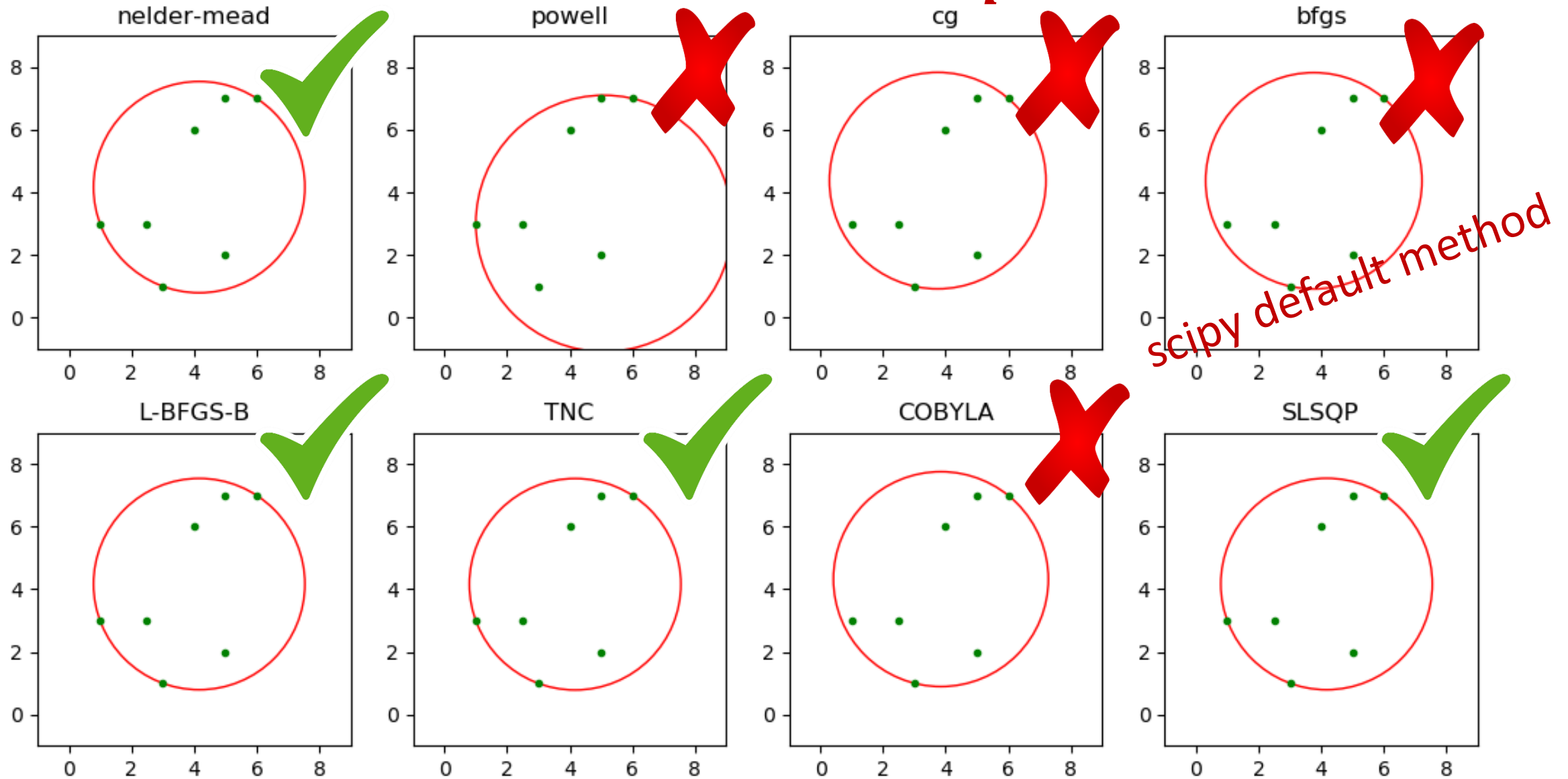
# Minimum enclosing circle - contour plot
plt.subplot(1, 2, 2)
plt.title('pyplot.contour')
plt.plot(trace_x, trace_y, '.-', color='darkblue')
plt.plot(points_x, points_y, 'o', color='darkgreen')
plt.plot(*center, 'o', c='red')
qcs = plt.contour(X, Y, Z, levels=30, cmap='plasma')
plt.clabel(qcs, inline=1, fontsize=8, fmt='%1f')

plt.suptitle('Maximum distance to an input point')
plt.tight_layout()
plt.show()
```

numpy
arrays



scipy.minimize $f(c) = \max_p |p - c|$



scipy.minimize $f(c) = \max_p |p - c|^2$ avoids $\sqrt{}$

