Visualizing Data

Lesson 6

Study the library matplotlib using this notebook
- fill up the line chart and scatter plot section

https://colab.research.google.com/drive/1zudOVQrWxYv-rMc9YlbShrHvUjlyipCm?usp=sharing

VISUALIZING RESULTS

- earlier saw examples of different orders of growth of procedures
- used graphs to provide an intuitive sense of differences
- example of leveraging an existing library, rather than writing procedures from scratch
- Python provides libraries for (among other topics):
  - graphing
  - numerical computation
  - stochastic computation
- want to explore idea of using existing library procedures to guide processing and exploration of data

USING PYLAB

- can import library into computing environment

```
import pylab as plt
```

  - allows me to reference any library procedure as `plt.<procName>`

- provides access to existing set of graphing/plotting procedures

- here will just show some simple examples; lots of additional information available in documentation associated with `pylab`

SIMPLE EXAMPLE

- basic function plots two lists as x and y values
  - other data structures more powerful, use lists to demonstrate
- first, let's generate some example data

```
mySamples = []
myLinear = []
myQuadratic = []
myCubic = []
myExponential = []

for i in range(0, 30):
    mySamples.append(i)
    myLinear.append(i)
    myQuadratic.append(i**2)
    myCubic.append(i**3)
    myExponential.append(1.5**i)
```

selected 1.5 to keep displays visible, more likely value for order of growth example would be 2

SIMPLE EXAMPLE

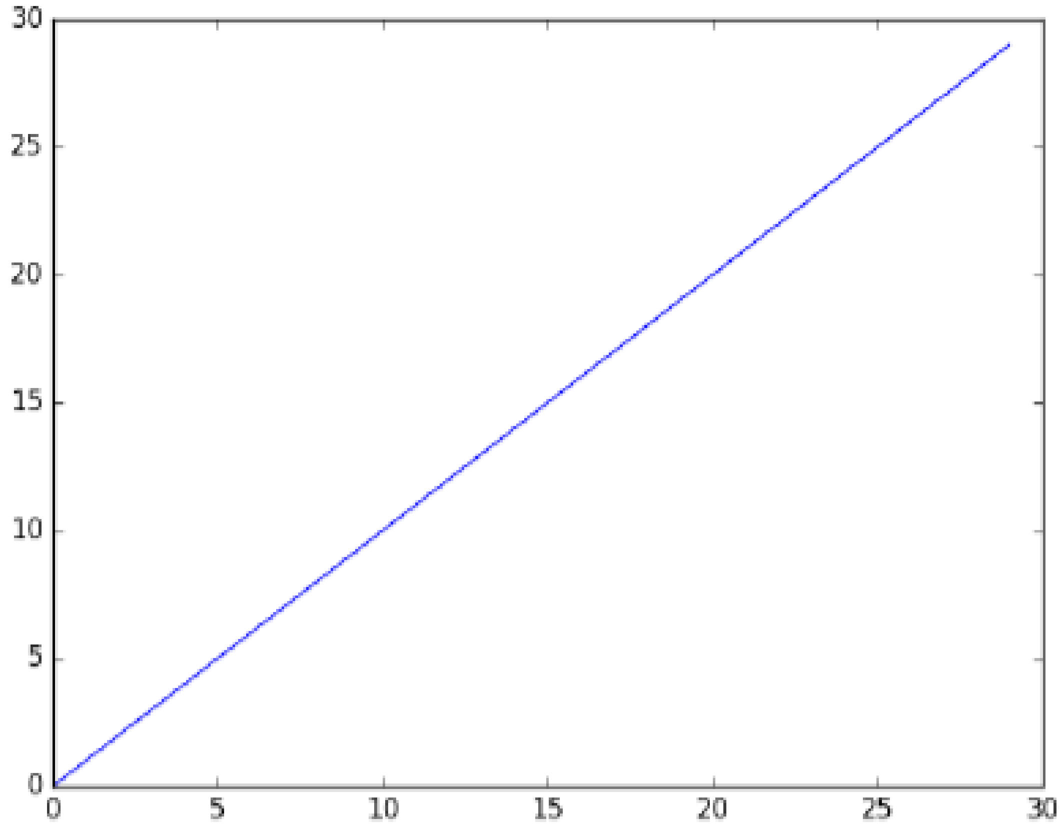
- to generate a plot, call

  X values / Y values

  ```
  plt.plot(mySamples, myLinear)
  ```

- arguments are lists of values (for now)
  - lists must be of the same length

EXAMPLE DISPLAY



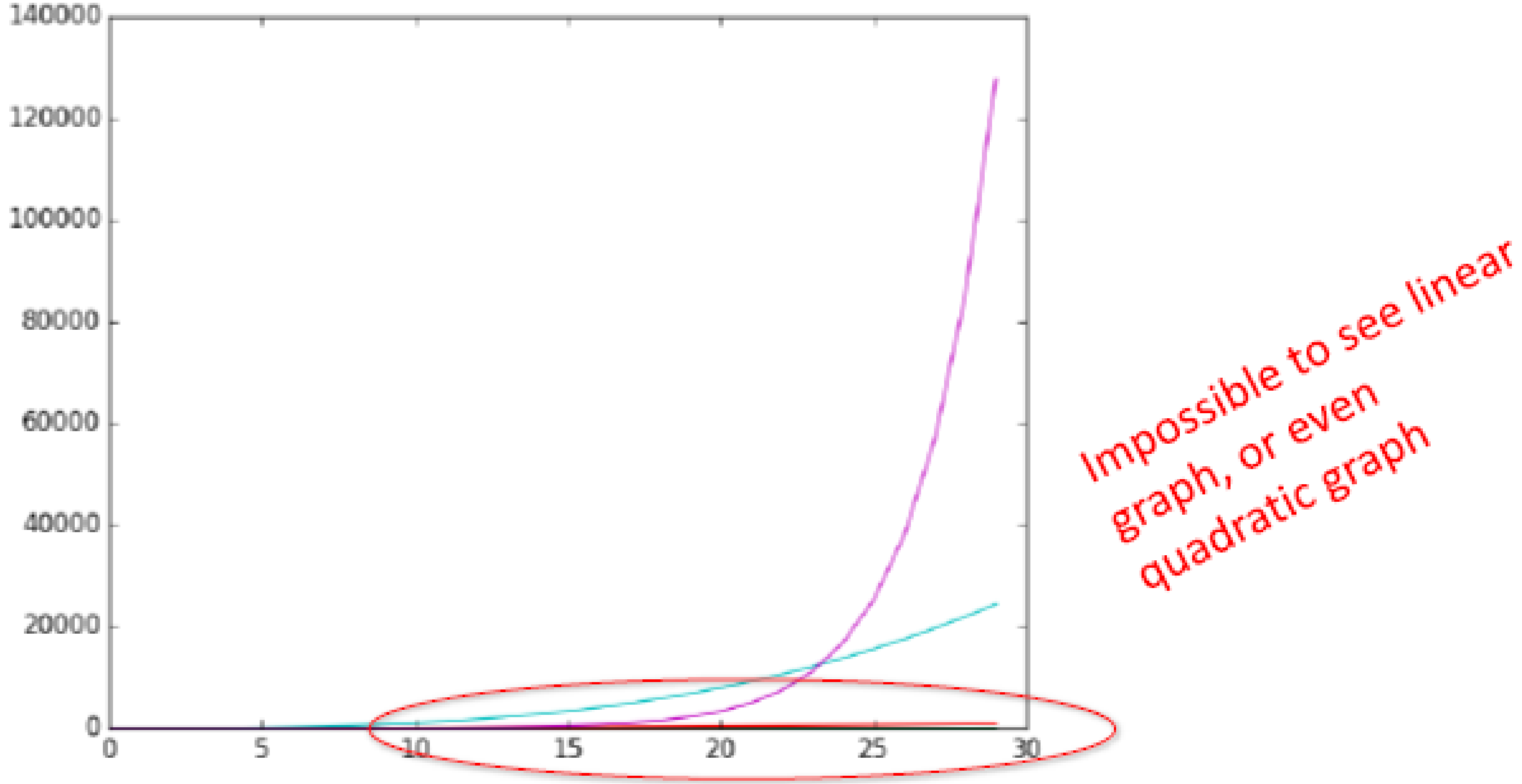

```
plt.plot(mySamples, myLinear)
```

OVERLAPPING DISPLAYS

- suppose we want to display all of the graphs of the different orders of growth
- we could just call:

```
plt.plot(mySamples, myLinear)
plt.plot(mySamples, myQuadratic)
plt.plot(mySamples, myCubic)
plt.plot(mySamples, myExponential)
```

EXAMPLE OVERLAY DISPLAY




```
plt.plot(mySamples, myLinear)
plt.plot(mySamples, myQuadratic)
```

```
plt.plot(mySamples, myCubic)
plt.plot(mySamples, myExponential)
```

OVERLAPPING DISPLAYS

- not very helpful, can't really see anything but the biggest of the plots because the scales are so different
- can we graph each one separately?
- call

```
plt.figure(<arg>)
```

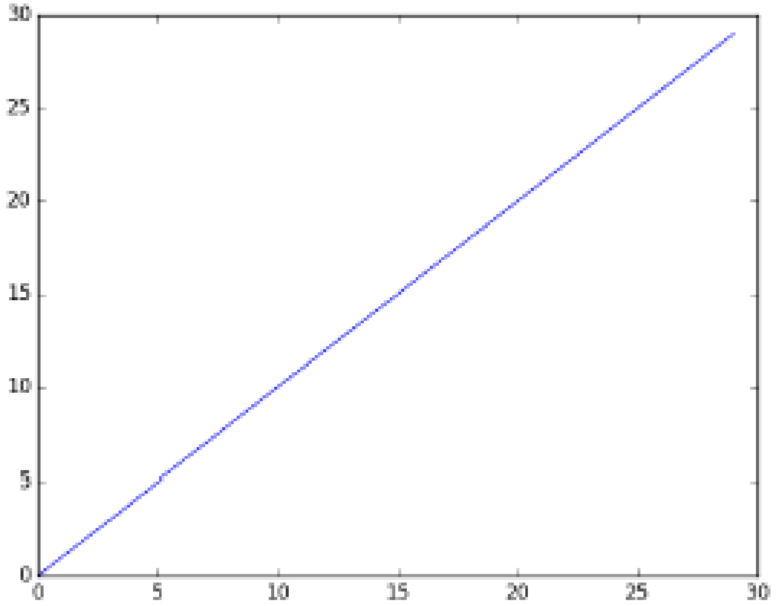
gives a name to this figure; allows us to reference for future use

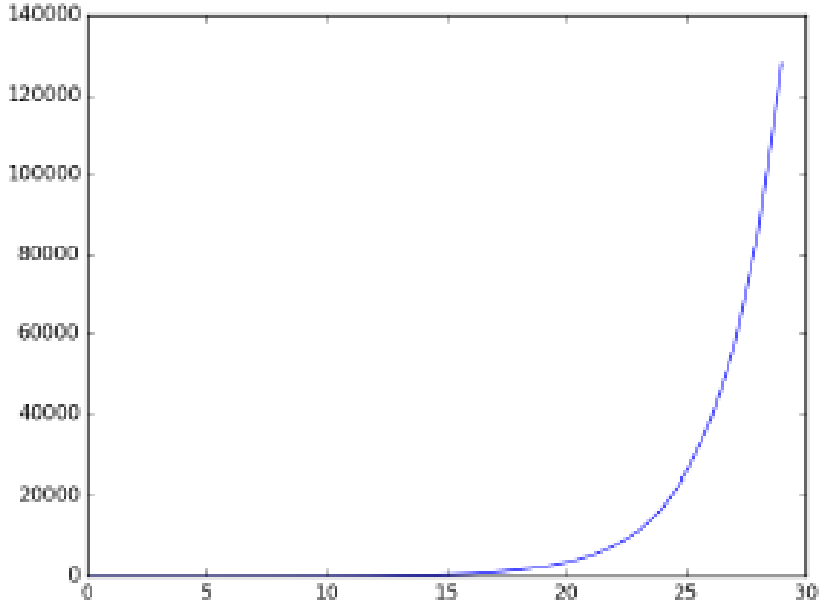
  - creates a new display with that name if one does not already exist
  - if a display with that name exists, reopens it for processing

EXAMPLE CODE

```
plt.figure('lin')
plt.plot(mySamples, myLinear)
plt.figure('quad')
plt.plot(mySamples, myQuadratic)
plt.figure('cube')
plt.plot(mySamples, myCubic)
plt.figure('expo')
plt.plot(mySamples, myExponential)
```

SEPARATE PLOTS






```
plt.figure('lin')
plt.plot(mySamples, myLinear)
```

```
plt.figure('expo')
plt.plot(mySamples,
myExponential)
```

PROVIDING LABELS

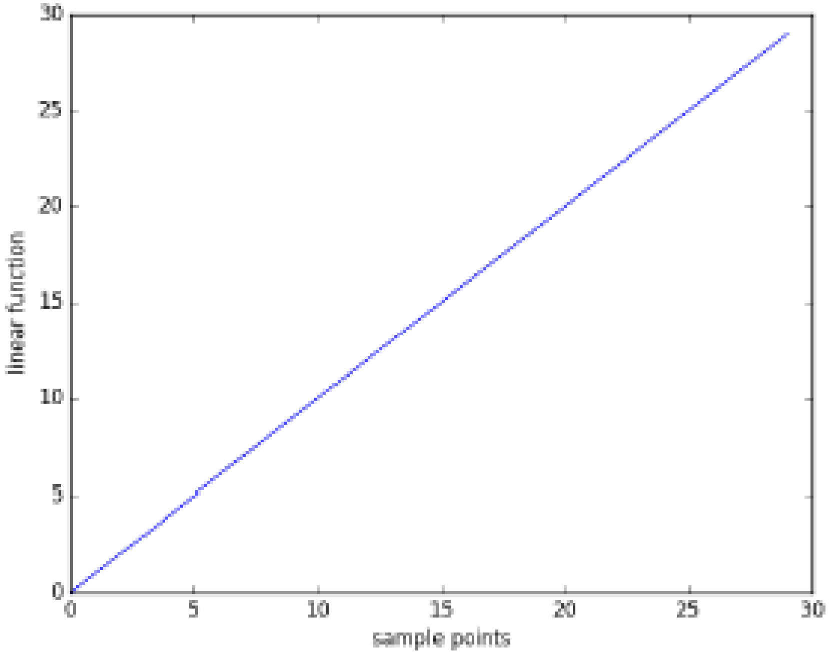
- Should really label the axes

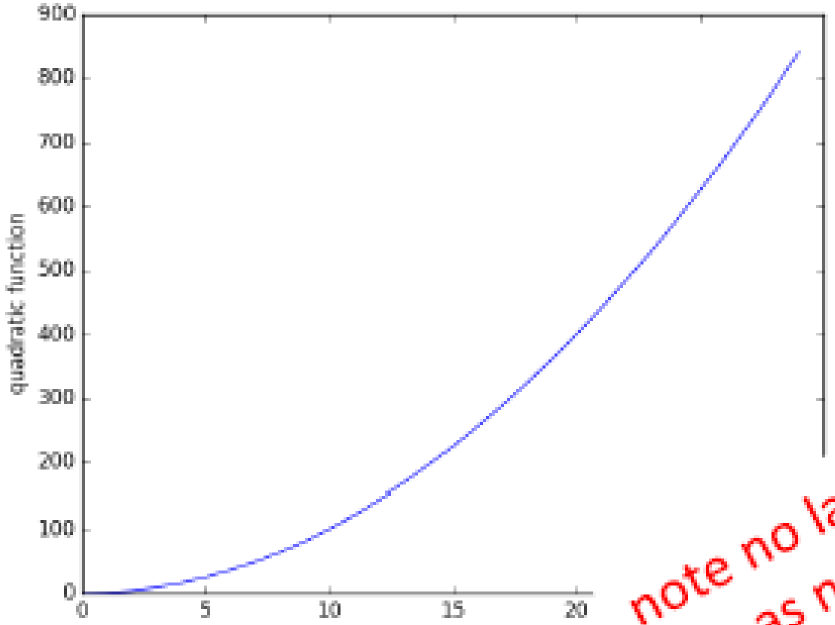
```
plt.figure('lin')
plt.xlabel('sample points')
plt.ylabel('linear function')
plt.plot(mySamples, myLinear)
plt.figure('quad')
plt.plot(mySamples, myQuadratic)
plt.figure('cube')
plt.plot(mySamples, myCubic)
plt.figure('expo')
plt.plot(mySamples, myExponential)
plt.figure('quad')
plt.ylabel('quadratic function')
```

functions to label axes

note you must make figure active before invoking labeling

LABELED AXES





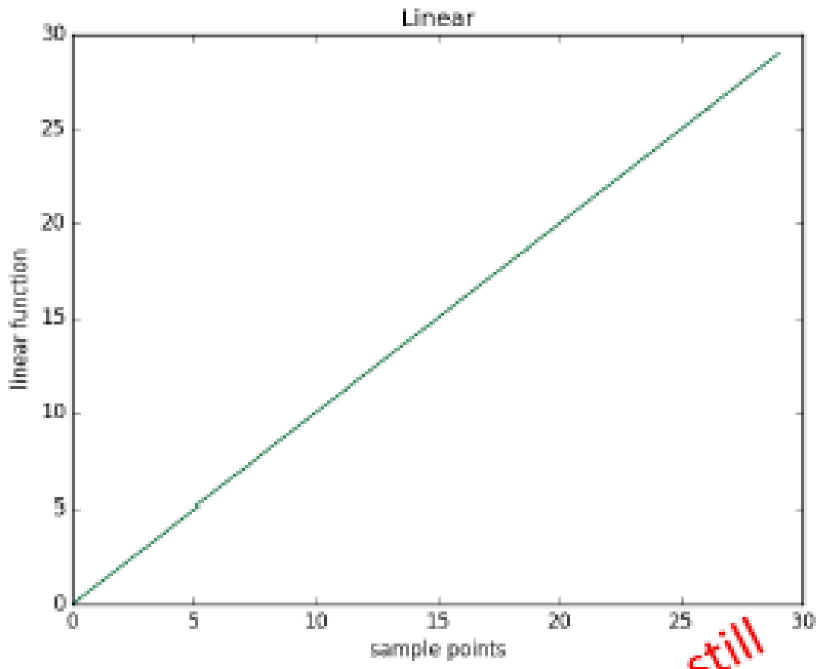
note no label on x axis as no invocation while that display was active

ADDING TITLES

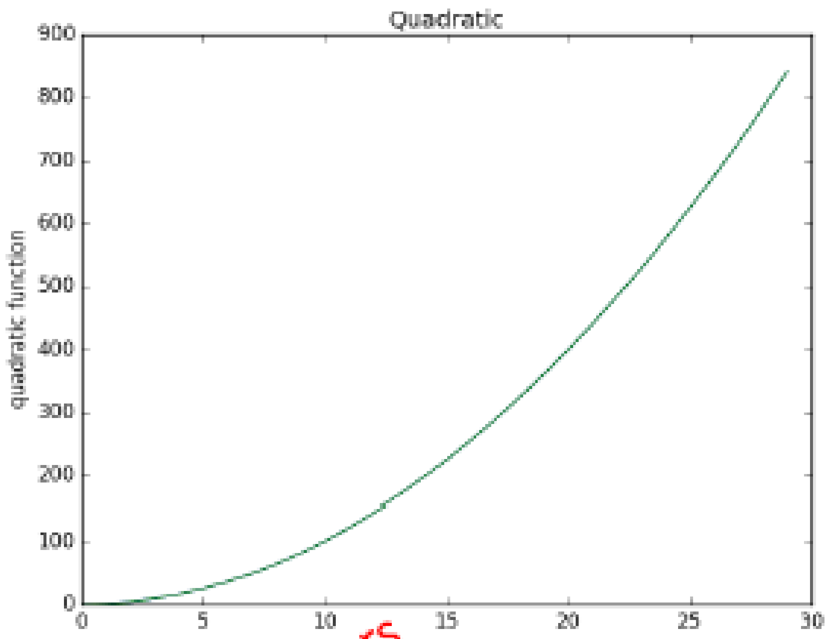
```
plt.figure('lin')
plt.plot(mySamples, myLinear)
plt.figure('quad')
plt.plot(mySamples, myQuadratic)
plt.figure('cube')
plt.plot(mySamples, myCubic)
plt.figure('expo')
plt.plot(mySamples, myExponential)
```

```
plt.figure('lin')
plt.title('Linear')
plt.figure('quad')
plt.title('Quadratic')
plt.figure('cube')
plt.title('Cubic')
plt.figure('expo')
plt.title('Exponential')
```

TITLED DISPLAYS



why are axes still labeled?



why are colors the same in the two plots?

CLEANING UP WINDOWS

- we are reusing a previously created display window
- need to clear it before redrawing

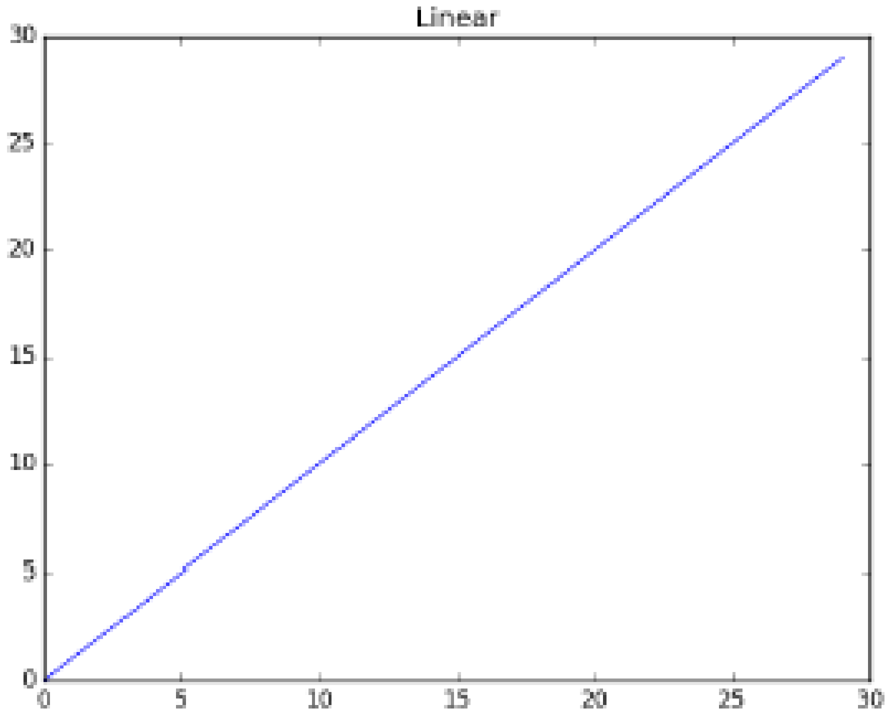
- because we are calling plot in a new version of a window, system starts with first choice of color (hence the same); we can control (see later)

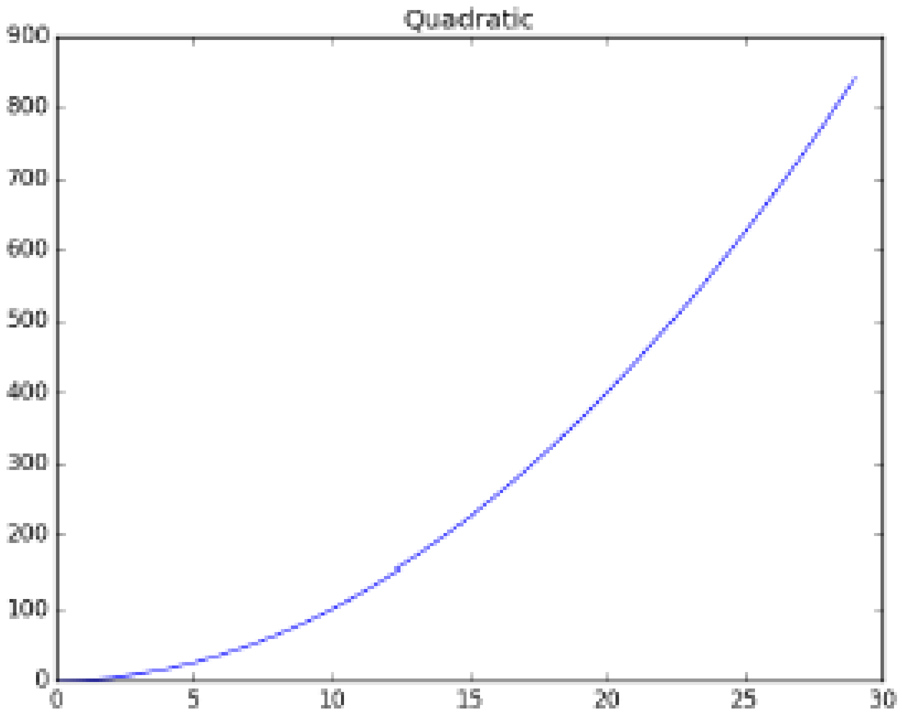
CLEANING WINDOWS

```
plt.figure('lin')
plt.clf()
plt.plot(mySamples, myLinear)
plt.figure('quad')
plt.clf()
plt.plot(mySamples, myQuadratic)
plt.figure('cube')
plt.clf()
plt.plot(mySamples, myCubic)
plt.figure('expo')
plt.clf()
plt.plot(mySamples, myExponential)
```

```
plt.figure('lin')
plt.title('Linear')
plt.figure('quad')
plt.title('Quadratic')
plt.figure('cube')
plt.title('Cubic')
plt.figure('expo')
plt.title('Exponential')
```

CLEARED DISPLAYS





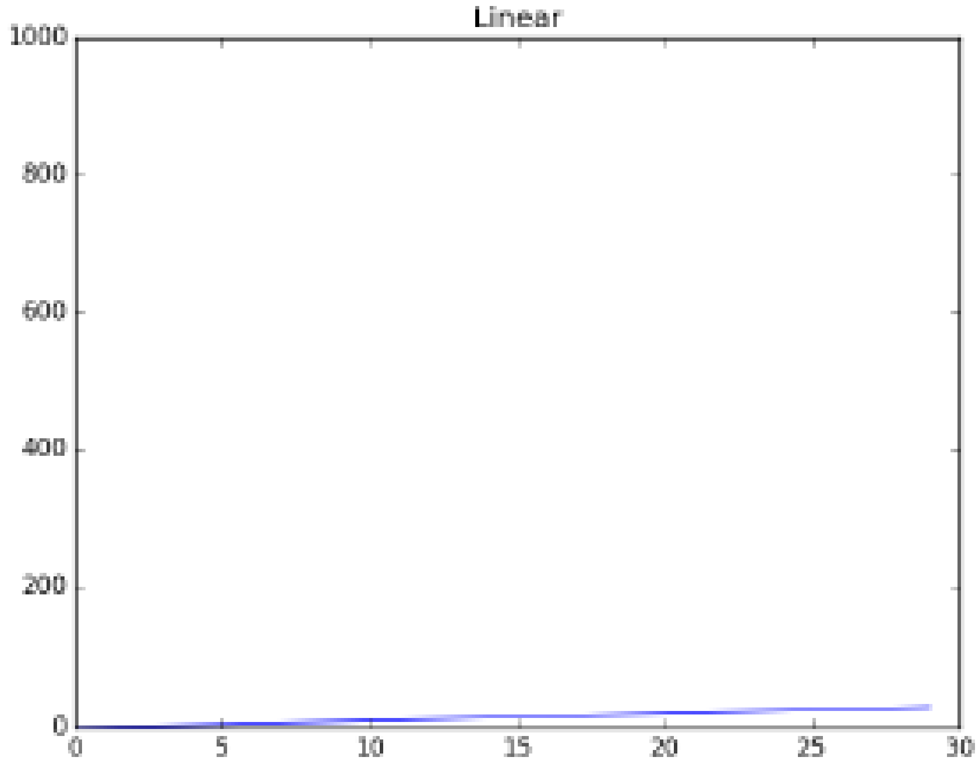
COMPARING RESULTS

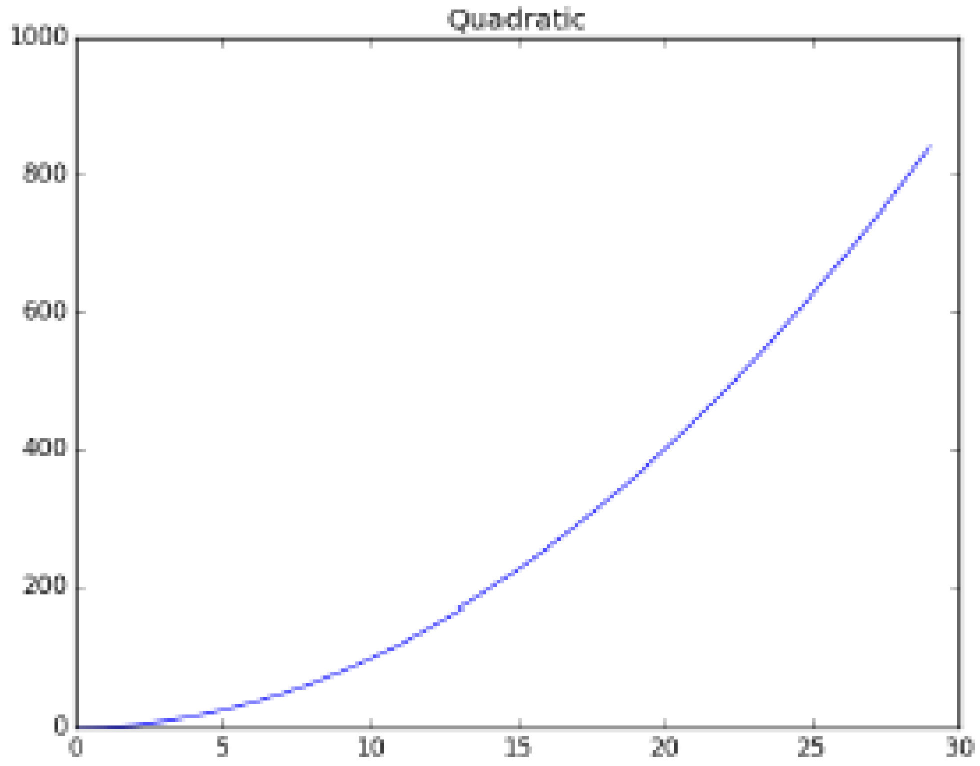
- now suppose we would like to compare different plots
- in particular, the scales on the graphs are very different
- one option is to explicitly set limits on the axis or axes
- a second option is to plot multiple functions on the same display

CHANGING LIMITS ON AXES

```
plt.figure('lin')
plt.clf()
plt.ylim(0,1000)
plt.plot(mySamples, myLinear)
plt.figure('quad')
plt.clf()
plt.ylim(0,1000)
plt.plot(mySamples, myQuadratic)
plt.figure('lin')
plt.title('Linear')
plt.figure('quad')
plt.title('Quadratic')
```

CHANGING LIMITS ON AXES





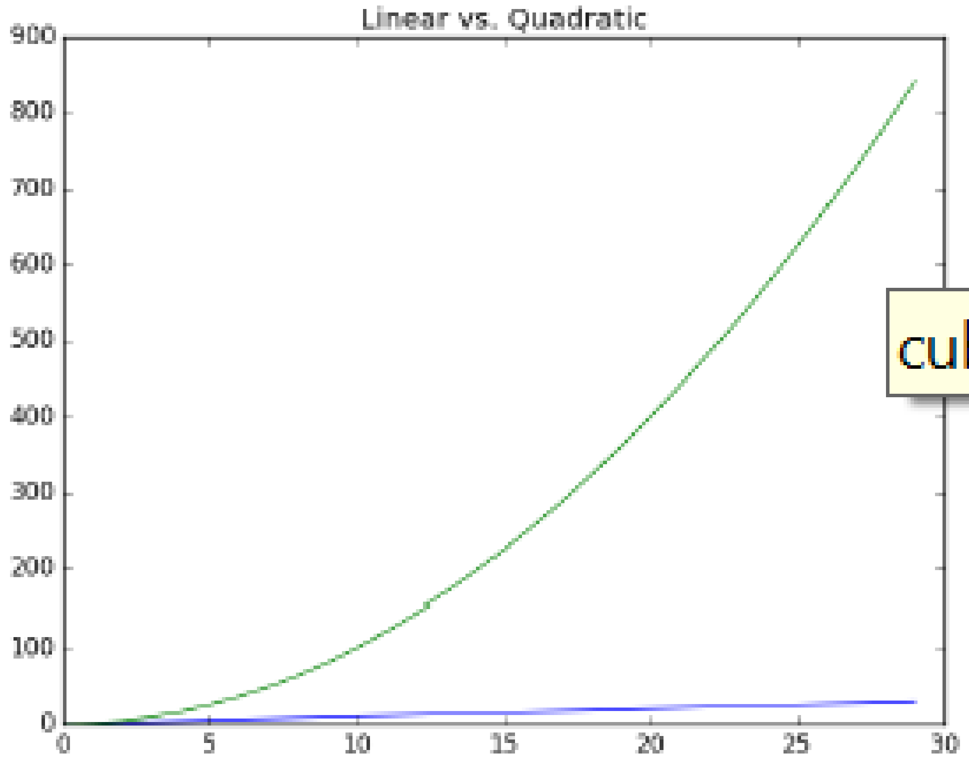
OVERLAYING PLOTS

```
plt.figure('lin quad')
plt.clf()
plt.plot(mySamples, myLinear)
plt.plot(mySamples, myQuadratic)

plt.figure('cube exp')
plt.clf()
plt.plot(mySamples, myCubic)
plt.plot(mySamples, myExponential)
plt.figure('lin quad')
plt.title('Linear vs. Quadratic')
plt.figure('cube exp')
plt.title('Cubic vs. Exponential')
```

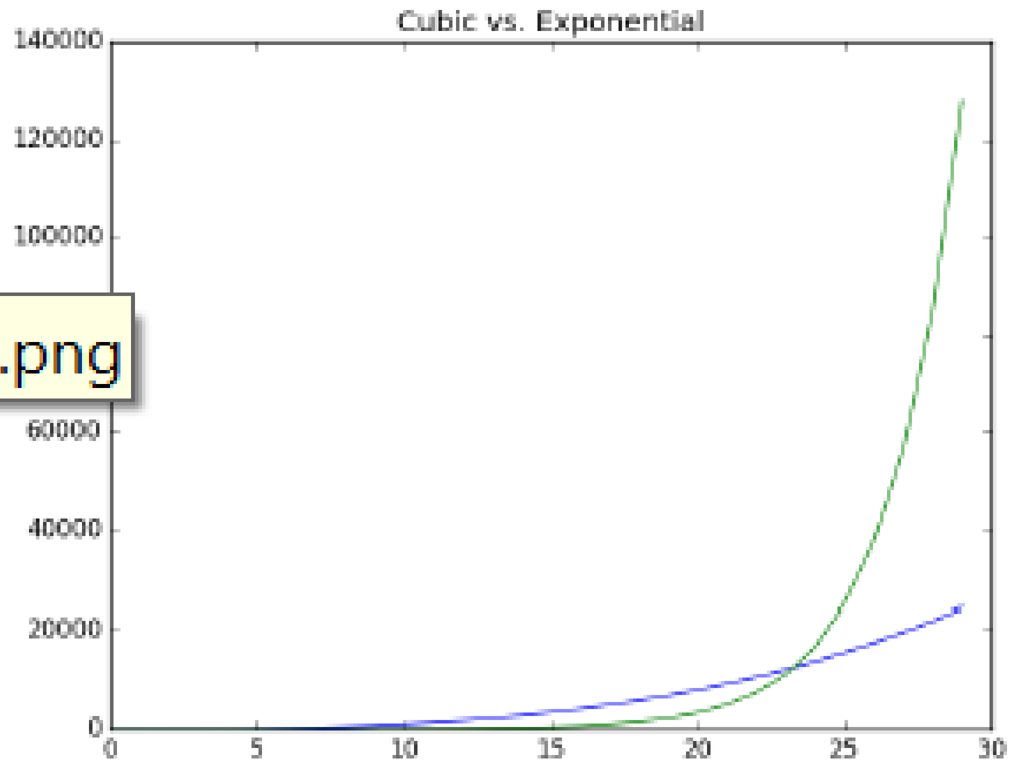
each pair of calls within the same active display window

each pair of calls within the same active display window

OVERLAYING PLOTS



cubeExp.png



ADDING MORE DOCUMENTATION

- can add a legend that identifies each plot

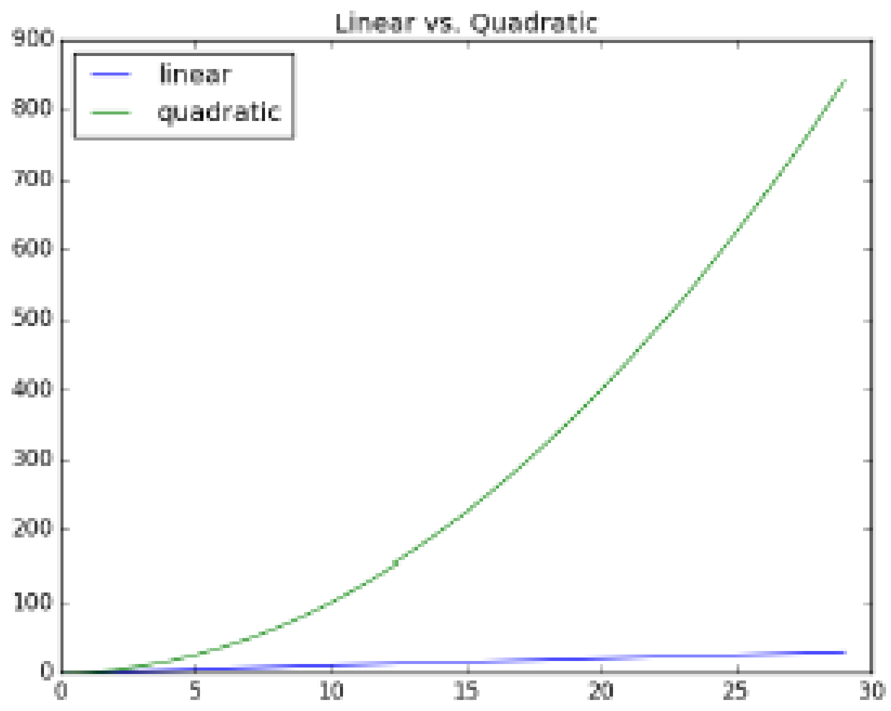
```
plt.figure('lin quad')
plt.clf()
plt.plot(mySamples, myLinear, label = 'linear')
plt.plot(mySamples, myQuadratic, label = 'quadratic')
plt.legend(loc = 'upper left')
plt.title('Linear vs. Quadratic')

plt.figure('cube exp')
plt.clf()
plt.plot(mySamples, myCubic, label = 'cubic')
plt.plot(mySamples, myExponential, label = 'exponential')
plt.legend()
plt.title('Cubic vs. Exponential')
```

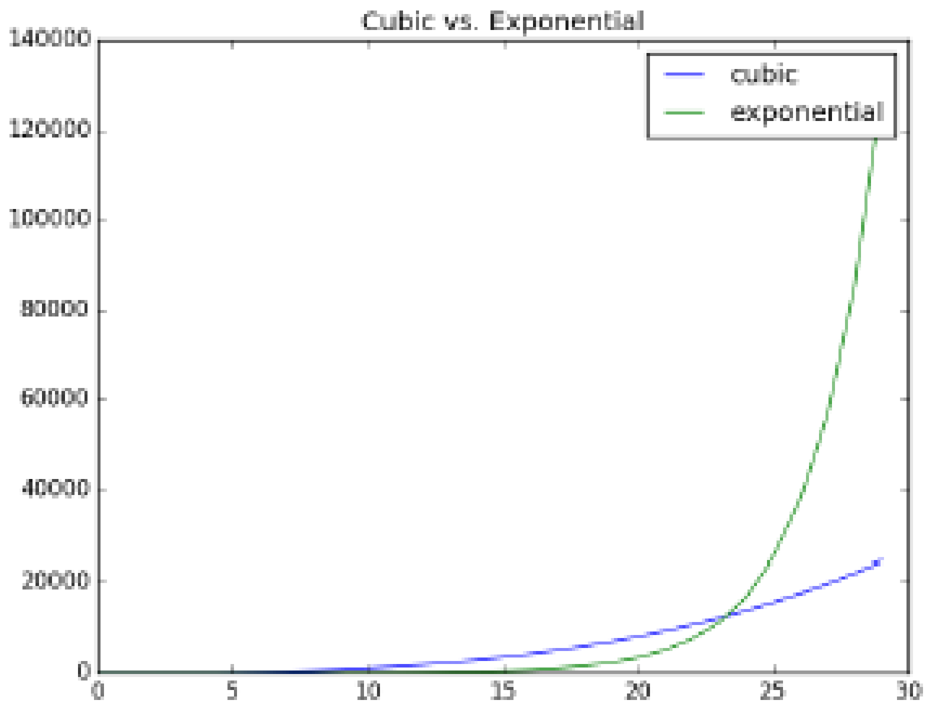
label each plot

can specify a location

can use best location

ADDING MORE DOCUMENTATION





CONTROLLING DISPLAY PARAMETERS

- now suppose we want to control details of the displays themselves
- examples:
  - changing color or style of data sets
  - changing width of lines or displays
  - using subplots

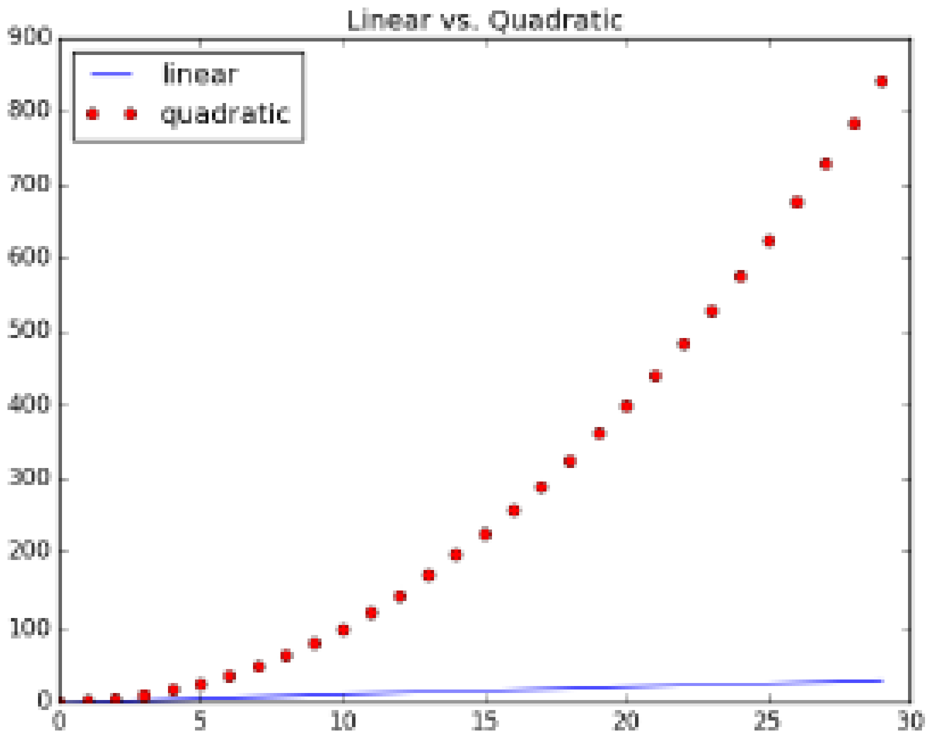
CHANGING DATA DISPLAY

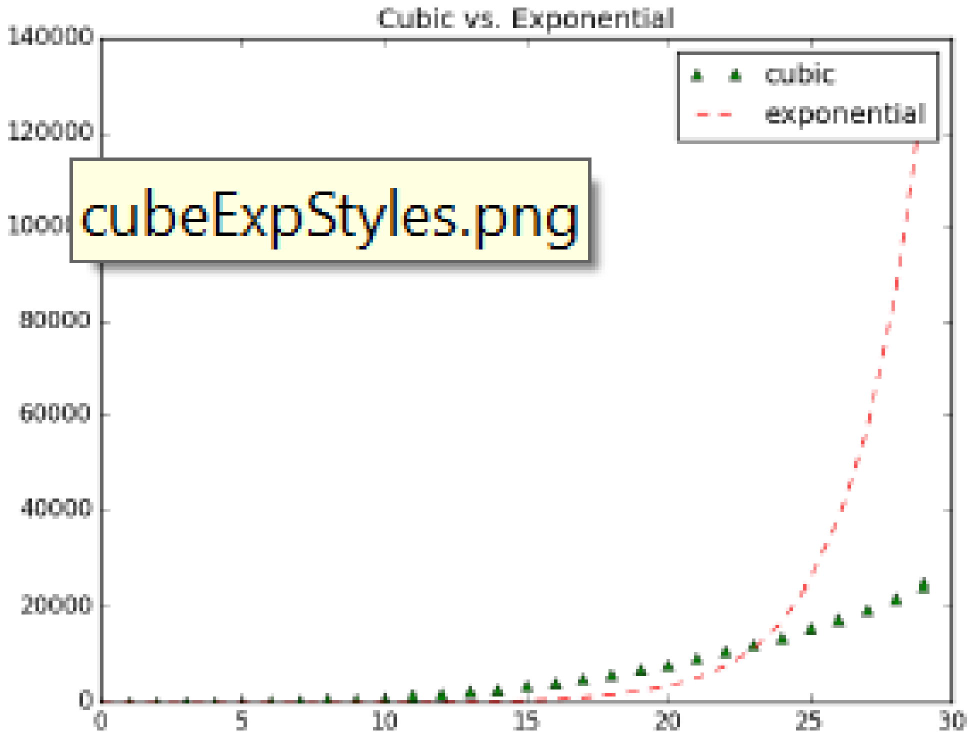
```
plt.figure('lin quad')
plt.clf()
plt.plot(mySamples, myLinear, 'b-', label = 'linear')
plt.plot(mySamples, myQuadratic,'ro', label = 'quadratic')
plt.legend(loc = 'upper left')
plt.title('Linear vs. Quadratic')

plt.figure('cube exp')
plt.clf()
plt.plot(mySamples, myCubic, 'g^', label = 'cubic')
plt.plot(mySamples, myExponential, 'r--',label = 'exponential')
plt.legend()
plt.title('Cubic vs. Exponential')
```

string specifies color and style

see documentation for choices of color and style

CHANGING DATA DISPLAY



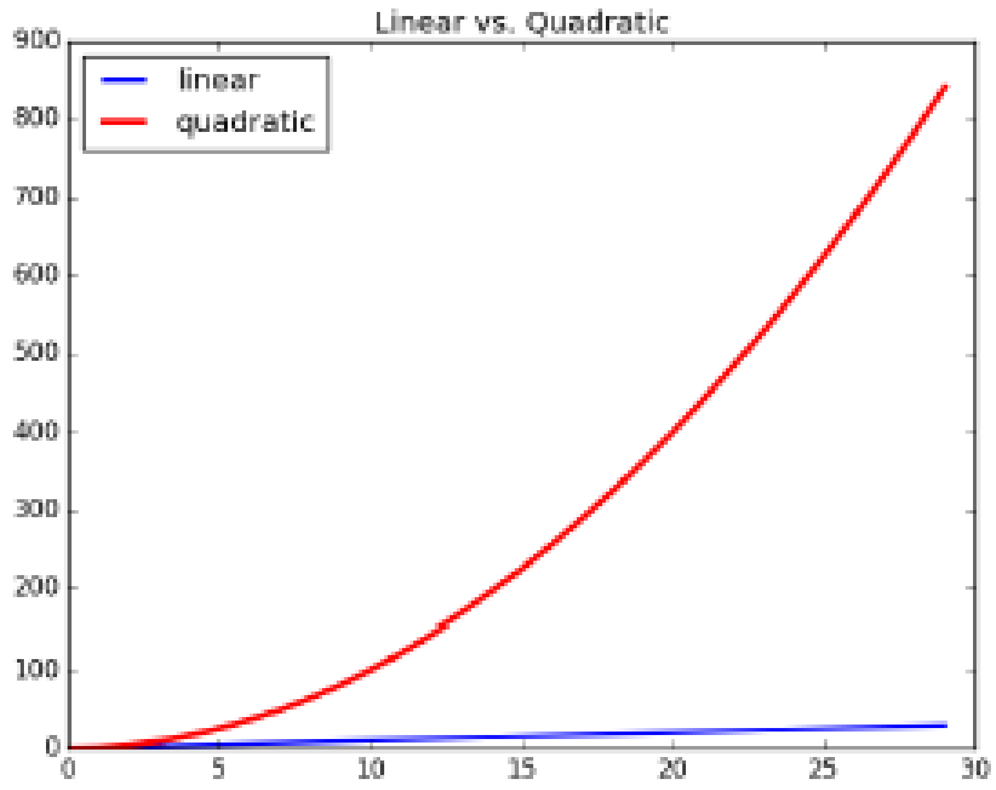


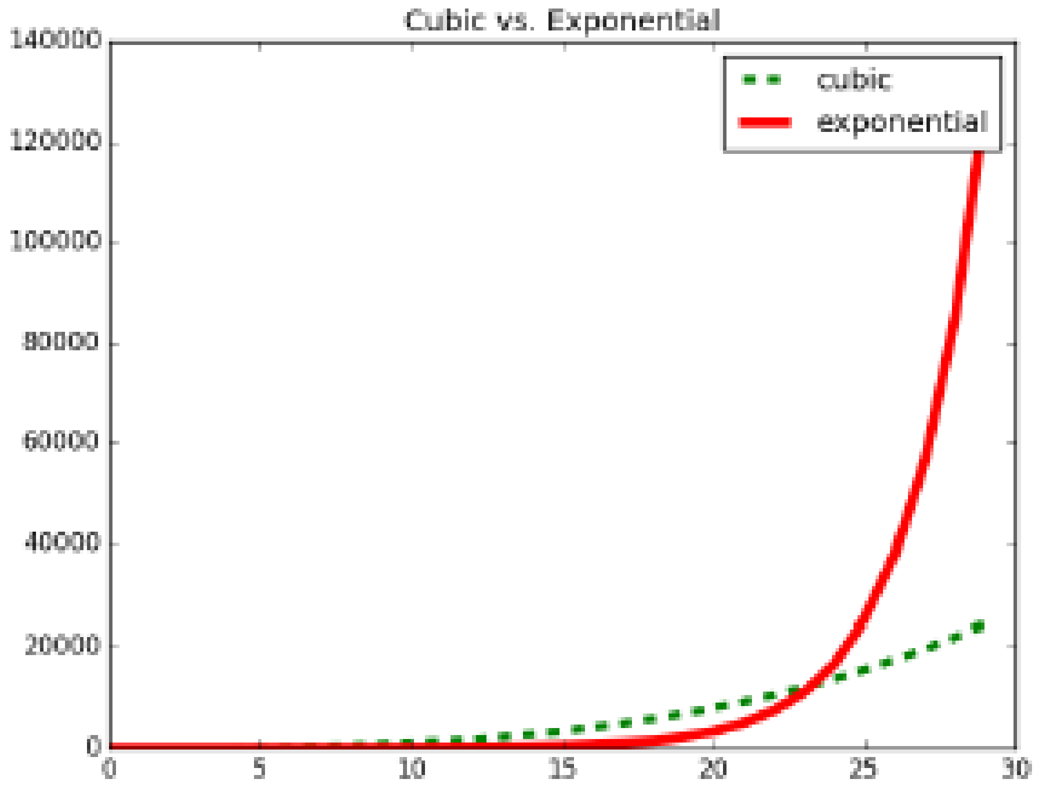
CHANGING DATA DISPLAY

```
plt.figure('lin quad')
plt.clf()
plt.plot(mySamples, myLinear, 'b-', label = 'linear', linewidth = 2.0)
plt.plot(mySamples, myQuadratic,'r', label = 'quadratic', linewidth = 3.0)
plt.legend(loc = 'upper left')
plt.title('Linear vs. Quadratic')

plt.figure('cube exp')
plt.clf()
plt.plot(mySamples, myCubic, 'g--', label = 'cubic', linewidth = 4.0)
plt.plot(mySamples, myExponential, 'r',label = 'exponential', linewidth = 5.0)
plt.legend()
plt.title('Cubic vs. Exponential')
```

keyword can change size of parameter

CHANGING DATA DISPLAY





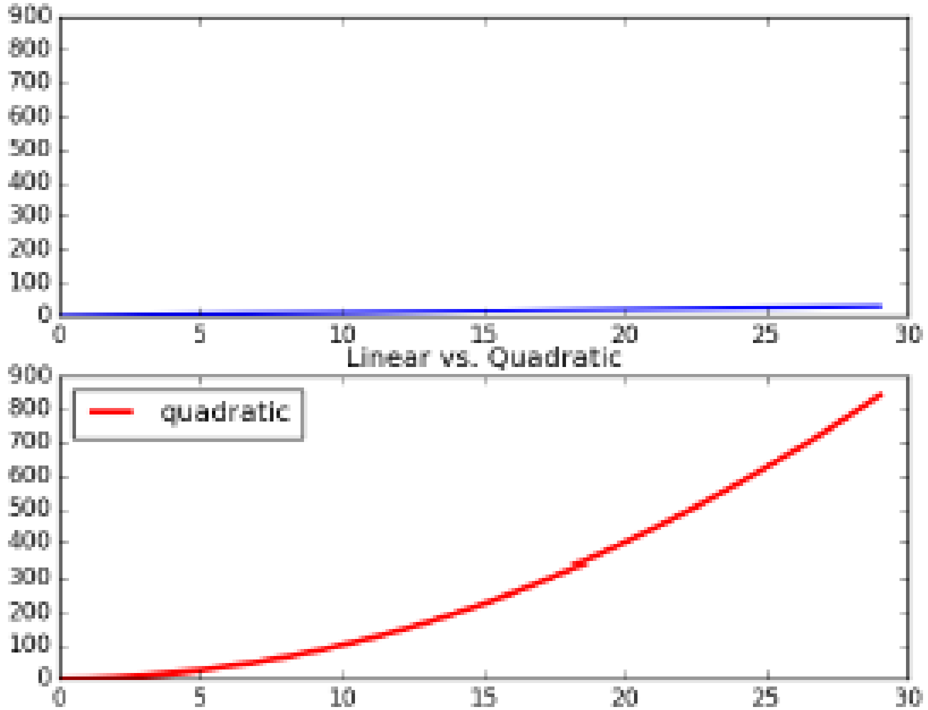
USING SUBPLOTS

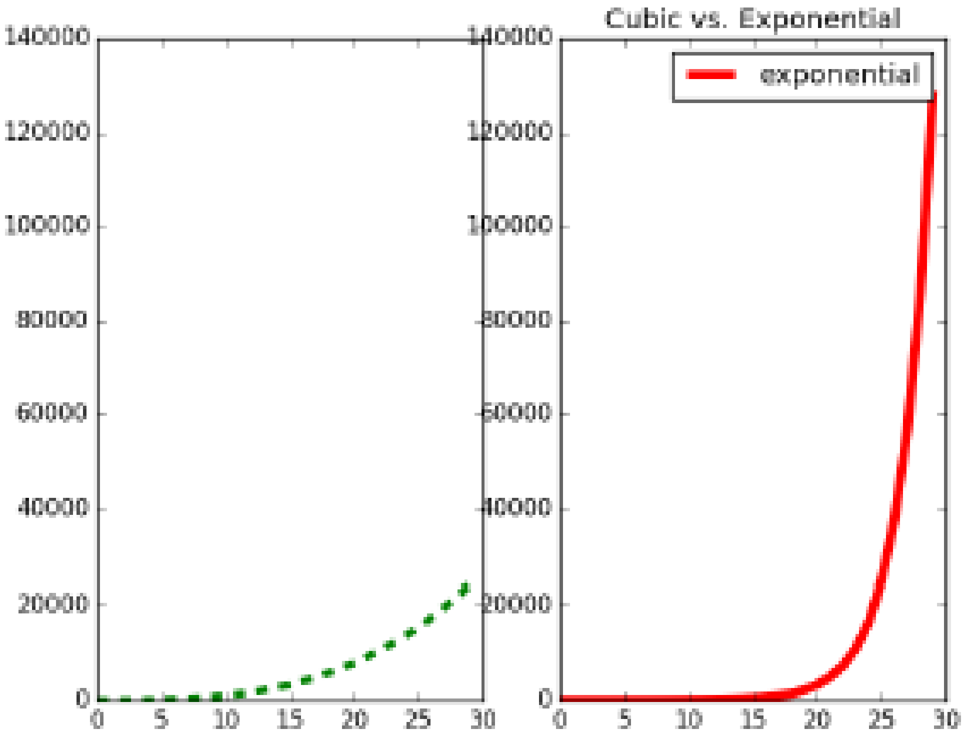
```
plt.figure('lin quad')
plt.clf()
plt.subplot(211)
plt.ylim(0,900)
plt.plot(mySamples, myLinear, 'b-', label = 'linear', linewidth = 2.0)
plt.subplot(212)
plt.ylim(0,900)
plt.plot(mySamples, myQuadratic, 'r', label = 'quadratic', linewidth = 3.0)
plt.legend(loc = 'upper left')
plt.title('Linear vs. Quadratic')

plt.figure('cube exp')
plt.clf()
plt.subplot(121)
plt.ylim(0, 140000)
plt.plot(mySamples, myCubic, 'g--', label = 'cubic', linewidth = 4.0)
plt.subplot(122)
plt.ylim(0, 140000)
plt.plot(mySamples, myExponential, 'r',label = 'exponential', linewidth = 5.0)
plt.legend()
plt.title('Cubic vs. Exponential')
```

arguments are number of rows & cols; and which location to use

set limit within each subplot

USING SUBPLOTS



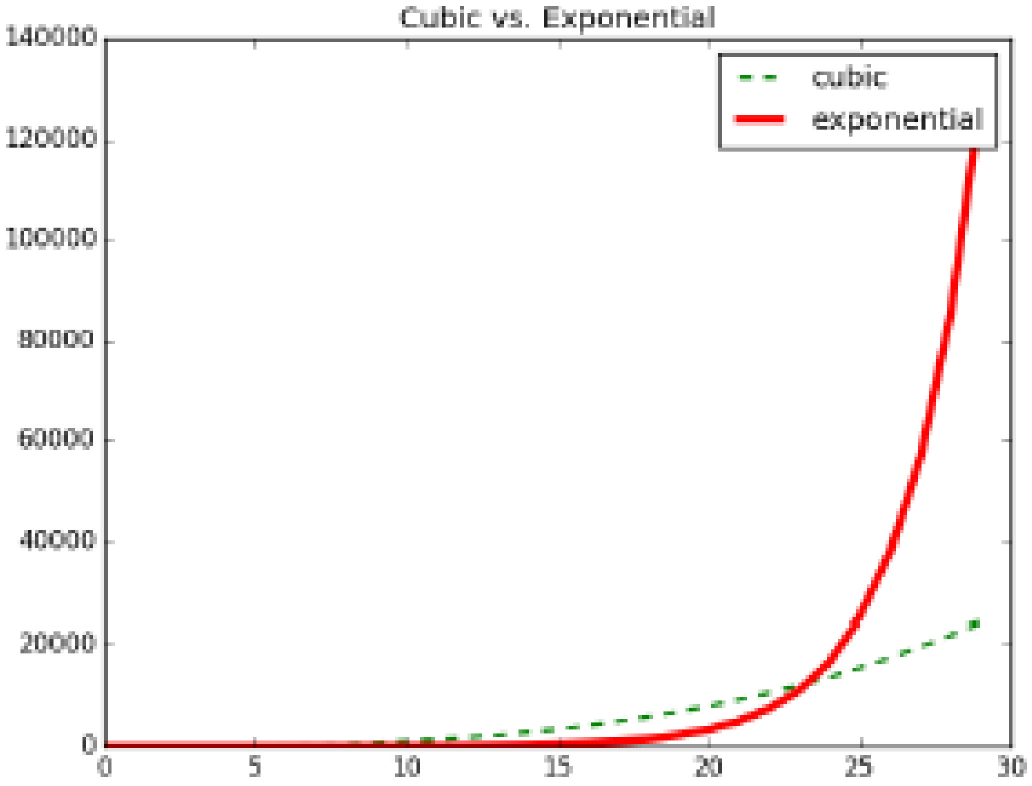


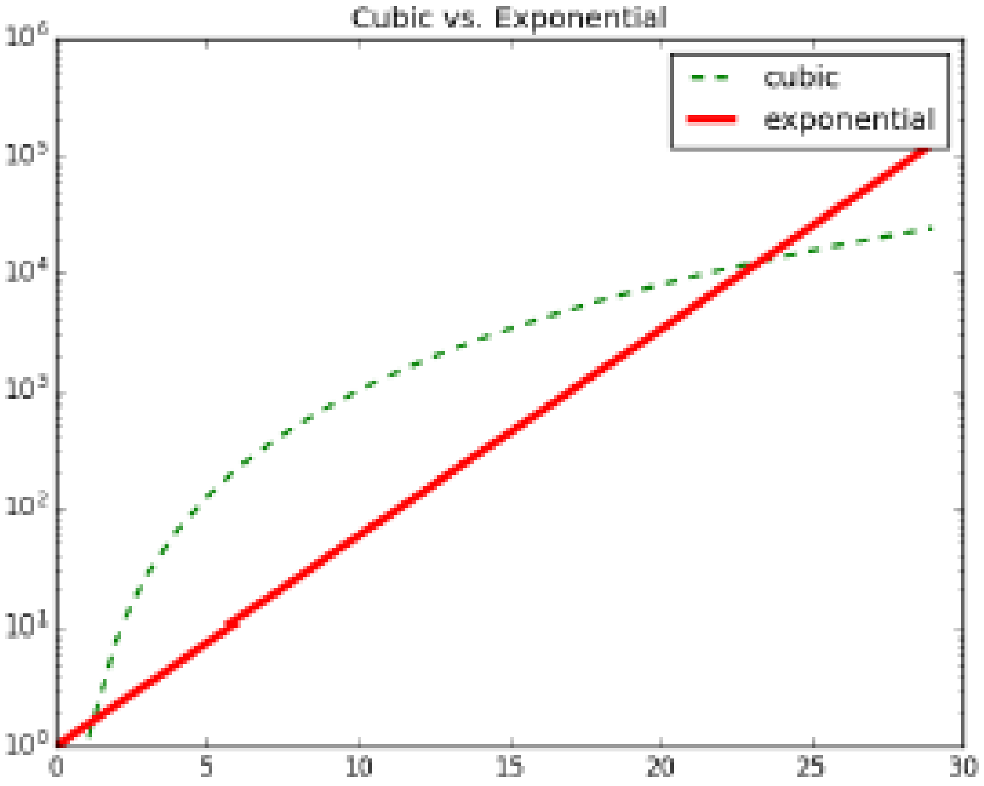
CHANGING SCALES

```
plt.figure('cube exp log')
plt.clf()
plt.plot(mySamples, myCubic, 'g--', label = 'cubic', linewidth = 2.0)
plt.plot(mySamples, myExponential, 'r',label = 'exponential', linewidth = 4.0)
plt.yscale('log')
plt.legend()
plt.title('Cubic vs. Exponential')

plt.figure('cube exp linear')
plt.clf()
plt.plot(mySamples, myCubic, 'g--', label = 'cubic', linewidth = 2.0)
plt.plot(mySamples, myExponential, 'r',label = 'exponential', linewidth = 4.0)
plt.legend()
plt.title('Cubic vs. Exponential')
```

argument specifies type of scaling

CHANGING SCALES





AN EXAMPLE

- want to explore how ability to visualize results can help guide computation
- simple example
  - planning for retirement
  - intend to save an amount `m` each month
  - expect to earn a percentage `r` of income on investments each month
  - want to explore how big a retirement fund will be compounded by time ready to retire

AN EXAMPLE: compound interest

```
def retire(monthly, rate, terms):
    savings = [0]
    base = [0]
    mRate = rate/12
    for i in range(terms):
        base += [i]
        savings += [savings[-1]*(1 + mRate) + monthly]
    return base, savings
```

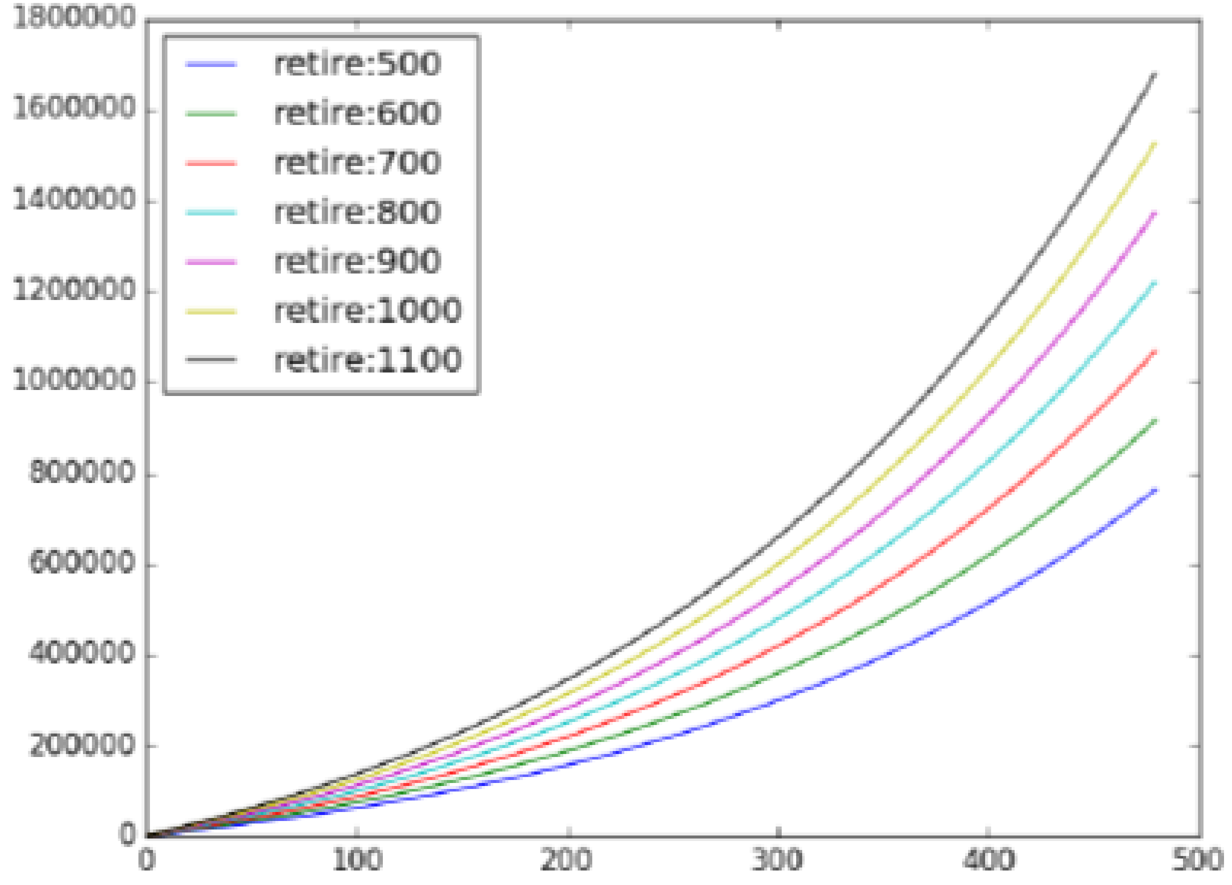
DISPLAYING RESULTS vs. MONTH

```
def displayRetireWMonthlies(monthlies, rate, terms):
    plt.figure('retireMonth')
    plt.clf()
    for monthly in monthlies:
        xvals, yvals = retire(monthly, rate, terms)
        plt.plot(xvals, yvals,
                 label = 'retire:'+str(monthly))
        plt.legend(loc = 'upper left')

displayRetireWMonthlies([500, 600, 700, 800, 900,
1000, 1100], .05, 40* 12)
```

clear frame so can reuse

put informative label on each

DISPLAYING RESULTS vs. MONTH



ANALYSIS vs. CONTRIBUTION

- can see impact of increasing monthly contribution
  - ranges from about 750K to 1.67M, as monthly savings ranges from $500 to $1100

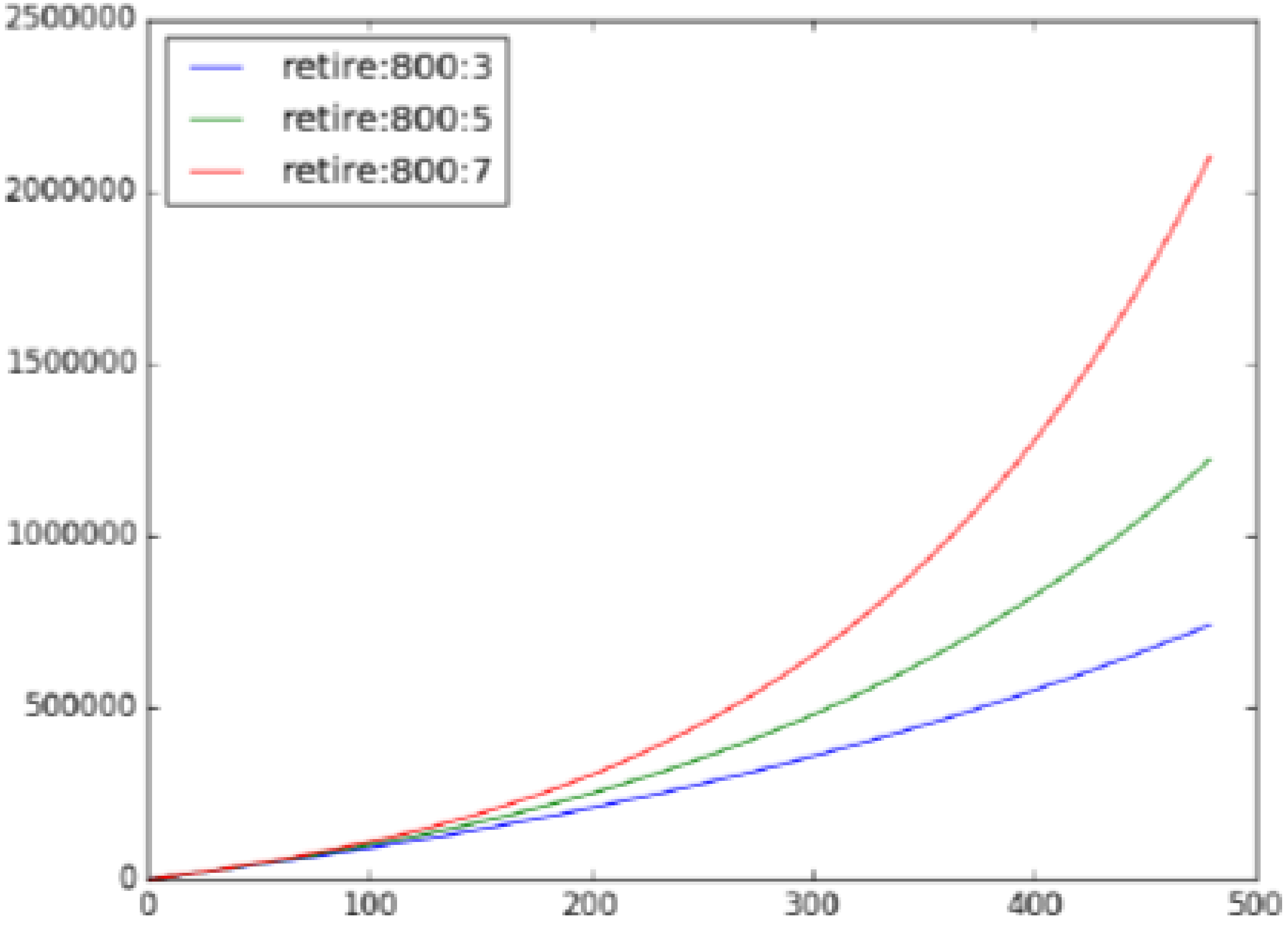
- what is effect of rate of growth of investments?

DISPLAYING RESULTS vs. RATE

```
def displayRetireWRates(month, rates, terms):
    plt.figure('retireRate')
    plt.clf()
    for rate in rates:
        xvals, yvals = retire(month, rate, terms)
        plt.plot(xvals, yvals,
                 label = 'retire:'+str(month)+ ':' + \
                         str(int(rate*100)))
        plt.legend(loc = 'upper left')

displayRetireWRates(800,[.03, .05, .07], 40*12)
```

put informative label on each

DISPLAYING RESULTS vs. RATE



ANALYSIS vs. RATE

- can also see impact of increasing expected rate of return on investments
  - ranges from about 600K to 2.1M, as rate goes from 3% to 7%

- what if we look at both effects together?

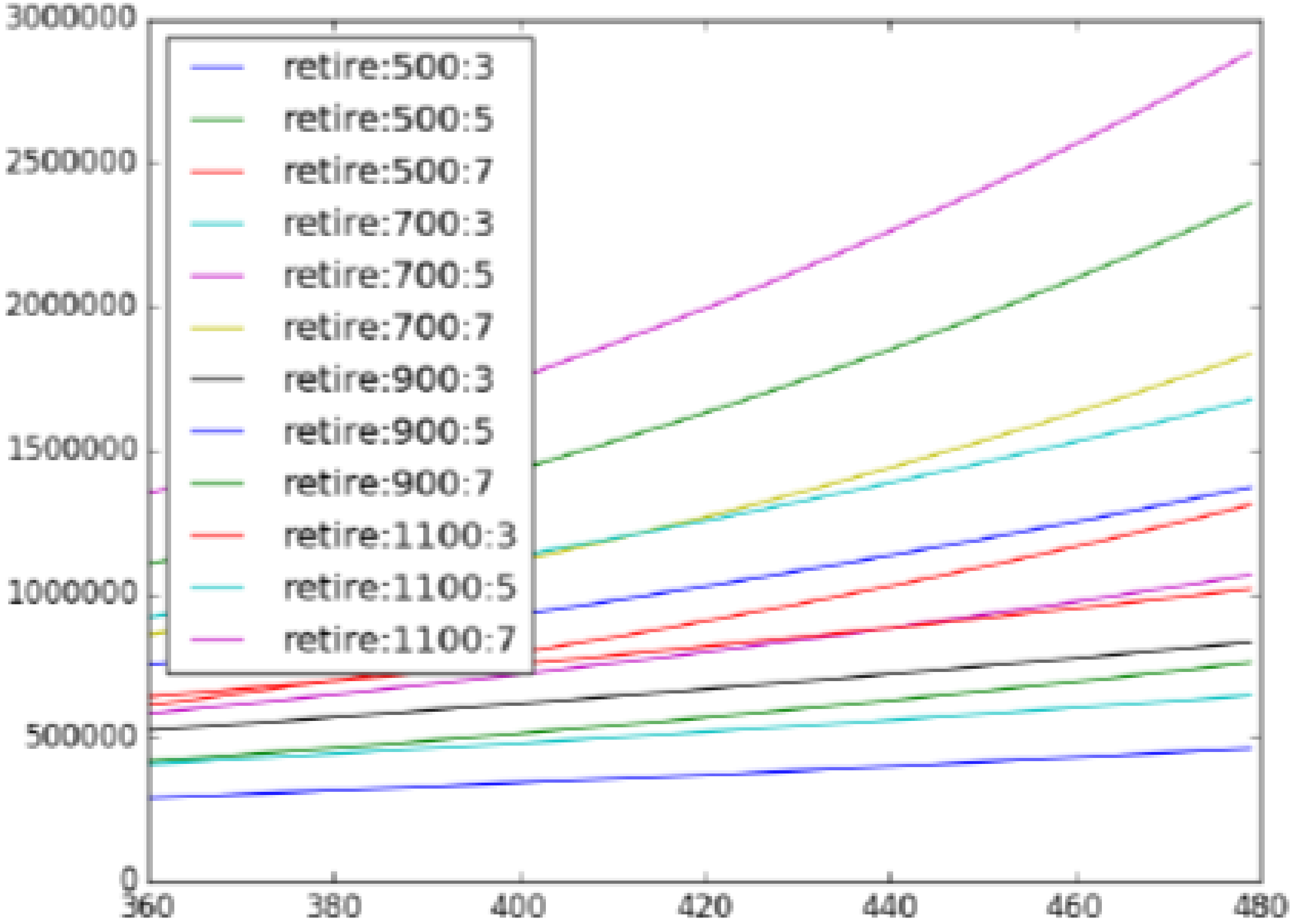
DISPLAYING RESULTS vs. BOTH

```
def displayRetireWMonthsAndRates(monthlies, rates, terms):
    plt.figure('retireBoth')
    plt.clf()
    plt.xlim(30*12, 40*12)
    for monthly in monthlies:
        for rate in rates:
            xvals, yvals = retire(monthly, rate, terms)
            plt.plot(xvals, yvals,
                     label = 'retire:'+str(monthly)+ ':' \
                             + str(int(rate*100)))
            plt.legend(loc = 'upper left')

displayRetireWMonthsAndRates([500, 700, 900, 1100],
                             [.03, .05, .07],
                             40*12)
```

focus on last stages of fund

put informative label on each

DISPLAYING RESULTS vs. BOTH



DISPLAYING RESULTS vs. BOTH

- hard to distinguish because of overlap of many graphs
- could just analyze separately
- but can also try to visually separate effects

DISPLAYING RESULTS vs. BOTH

```
def displayRetireWMonthsAndRates(monthlies, rates, terms):
    plt.figure('retireBoth')
    plt.clf()
    plt.xlim(30*12, 40*12)
    monthLabels = ['r', 'b', 'g', 'k']
    rateLabels = ['-', 'o', '-']
    for i in range(len(monthlies)):
        monthly = monthlies[i]
        monthLabel = monthLabels[i%len(monthLabels)]
        for j in range(len(rates)):
            rate = rates[j]
            rateLabel = rateLabels[j%len(rateLabels)]
            xvals, yvals = retire(monthly, rate, terms)
            plt.plot(xvals, yvals,
                     monthLabel+rateLabel,
                     label = 'retire:'+str(monthly)+ ':' \
                             + str(int(rate*100)))
            plt.legend(loc = 'upper left')

displayRetireWMonthsAndRates([500, 700, 900, 1100], [.03, .05, .07],
                             40*12)
```

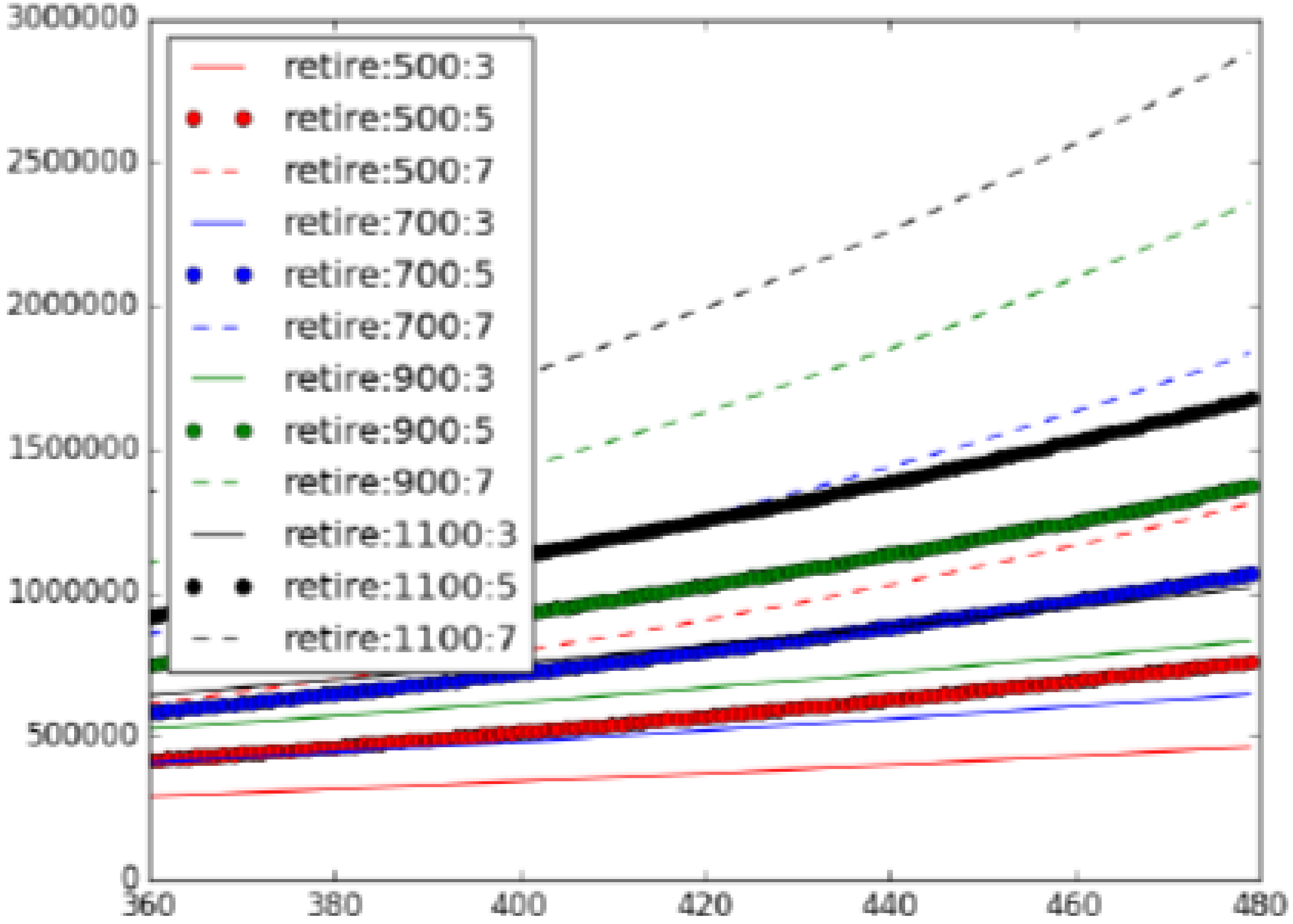
create sets of labels

pick new label for each month choice

pick new label for each rate choice

create label for plot

DISPLAYING RESULTS vs. BOTH



DISPLAYING RESULTS vs. BOTH

- now easier to see grouping of plots
  - color encodes monthly contribute
  - format (solid, circle, dashed) encodes growth rate of investments

- interaction with plotting routines and computations allows us to explore data
  - change display range to zero in on particular areas of interest
  - change sets of values and visualize effect – then guides new choice of values to explore
  - change display parameters to highlight clustering of plots by parameter

Quiz 6 – 15 mins

- Start 1500
- you can refer to Readings
- raise your hand if you want to submit early
- you are to wait patiently for everyone to submit their Quiz or when time runs out
- you can type out the code in Lesson Slides 7 to follow with the lesson

Stochastic Thinking and Random Walks

Lesson 7

The World is Hard to Understand

- Uncertainty is uncomfortable
- But certainty is usually unjustified

Newtonian Mechanics

- Every effect has a cause
- The world can be understood causally

Copenhagen Doctrine

- Copenhagen Doctrine (Bohr and Heisenberg) of causal nondeterminism
  - At its most fundamental level, the behavior of the physical world cannot be predicted.
  - Fine to make statements of the form “x is highly likely to
  - occur,” but not of the form “x is certain to occur.”
- Einstein and Schrödinger objected
  - “God does not play dice.” -- Albert Einstein

Does It Really Matter

Did the flips yield

2 heads

2 tails

1 head and 1 tail?

The Moral

- The world may or may not be inherently unpredictable
- But our lack of knowledge does not allow us to make accurate predictions
- Therefore we might as well treat the world as inherently unpredictable
- Predictive nondeterminism

Stochastic Processes

- An ongoing process where the next state might depend on both the previous states and some random element

```
def rollDie():
    """returns an int between 1 and 6"""
```

```
def rollDie():
    """returns a randomly chosen int
       between 1 and 6"""
```

Implementing a Random Process

```
import random

def rollDie():
    """returns a random int between 1 and 6"""
    return random.choice([1,2,3,4,5,6])

def testRoll(n = 10):
    result = ''
    for i in range(n):
        result = result + str(rollDie())
    print(result)
```

Probability of Various Results

- Consider `testRoll(5)`
- How probable is the output 11111?

Probability Is About Counting

- Count the number of possible events
- Count the number of events that have the property of interest
- Divide one by the other
- Probability of 11111?
    - 11111, 11112, 11113, …, 11121, 11122, …, 66666
    - 1/(6**5)
    - ~0.0001286

Three Basic Facts About Probability

- Probabilities are always in the range 0 to 1. 0 if impossible, and 1 if guaranteed.
- If the probability of an event occurring is p, the probability of it not occurring must be
- When events are independent of each other, the probability of all of the events occurring is equal to a product of the probabilities of each of the events occurring.

Independence

- Two events are independent if the outcome of one event has no influence on the outcome of the other
- Independence should not be taken for granted

Winning and losing probability of two teams out of many teams.

A Simulation of Die Rolling

```
def runSim(goal, numTrials, txt):
    total = 0
    for i in range(numTrials):
        result = ''
        for j in range(len(goal)):
            result += str(rollDie())
        if result == goal:
            total += 1
    print('Actual probability of', txt, '=',
          round(1/(6**len(goal)), 8))
    estProbability = round(total/numTrials, 8)
    print('Estimated Probability of', txt, '=',
          round(estProbability, 8))

runSim('11111', 1000, '11111')
```

Output of Simulation

- Actual probability = 0.0001286
- Estimated Probability = 0.0
- Actual probability = 0.0001286
- Estimated Probability = 0.0

- How did I know that this is what would get printed?
- Why did simulation give me the wrong answer?

Let’s try 1,000,000 trials

Morals

- Moral 1: It takes a lot of trials to get a good estimate of the frequency of occurrence of a rare event. We'll talk lots more in later lectures about how to know when we have enough trials.
- Moral 2: One should not confuse the sample probability with the actual probability
- Moral 3: There was really no need to do this by simulation, since there is a perfectly good closed form answer. We will see many examples where this is not true.
- But simulations are often useful.

The Birthday Problem

- What's the probability of at least two people in a group having the same birthday
- If there are 367 people in the group?
- What about smaller numbers?
- If we assume that each birthdate is equally likely
  - $1 - \frac{366!}{366^N * (366-N)!}$
- Without this assumption, VERY complicated

Try to write a program that simulates and get an approximation

Approximating Using a Simulation

```
def sameDate(numPeople, numSame):
    possibleDates = range(366)
    birthdays = [0]*366
    for p in range(numPeople):
        birthDate = random.choice(possibleDates)
        birthdays[birthDate] += 1
    return max(birthdays) >= numSame
```

Approximating Using a Simulation

```
def birthdayProb(numPeople, numSame, numTrials):
    numHits = 0
    for t in range(numTrials):
        if sameDate(numPeople, numSame):
            numHits += 1
    return numHits/numTrials

for numPeople in [10, 20, 40, 100]:
    print('For', numPeople,
          'est. prob. of a shared birthday is',
          birthdayProb(numPeople, 2, 10000))
    numerator = math.factorial(366)
    denom = (366**numPeople)*math.factorial(366-numPeople)
    print('Actual prob. for N = 100 =',
          1 - numerator/denom)
```

Suppose we want the probability of 3 people sharing

Why 3 Is Much Harder Mathematically

- For 2 the complementary problem is “all birthdays distinct”
- For 3 people, the complementary problem is a complicated disjunct
  - All birthdays distinct or
  - One pair and rest distinct or
  - Two pairs and rest distinct or
  - …
- But changing the simulation is dead easy

But all birthdays are not equally likely.

Another Win for Simulation

- Adjusting analytic model a pain
- Adjusting simulation model easy

```
def sameDate(numPeople, numSame):
    possibleDates = 4*list(range(0, 57)) + [58]\
                    + 4*list(range(59, 366))\
                    + 4*list(range(180, 270))
    birthdays = [0]*366
    for p in range(numPeople):
        birthDate = random.choice(possibleDates)
        birthdays[birthDate] += 1
    return max(birthdays) >= numSame
```

Simulation Models

- A description of computations that provide useful information about the possible behaviors of the system being modeled
- Descriptive, not prescriptive
- Only an approximation to reality
- "All models are wrong, but some are useful." – George Box

Simulations Are Used a Lot

- To model systems that are mathematically intractable
- To extract useful intermediate results
- Lend themselves to development by successive refinement and “what if” questions
- Start by simulating random walks