Lecture 5: Random Walks

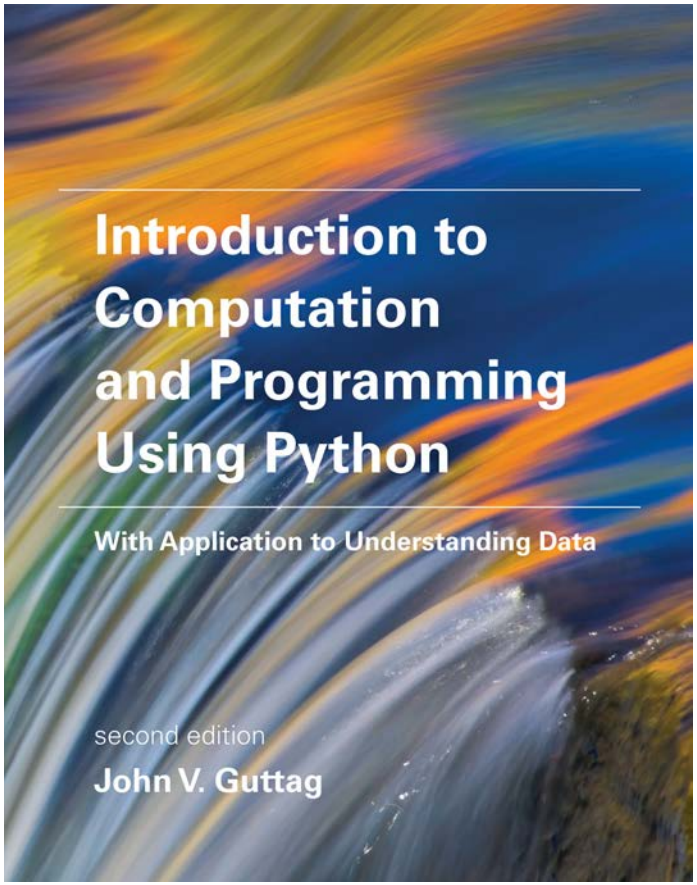
(download slides and .py files from Stellar to follow along)

John Guttag

MIT Department of Electrical Engineering and Computer Science

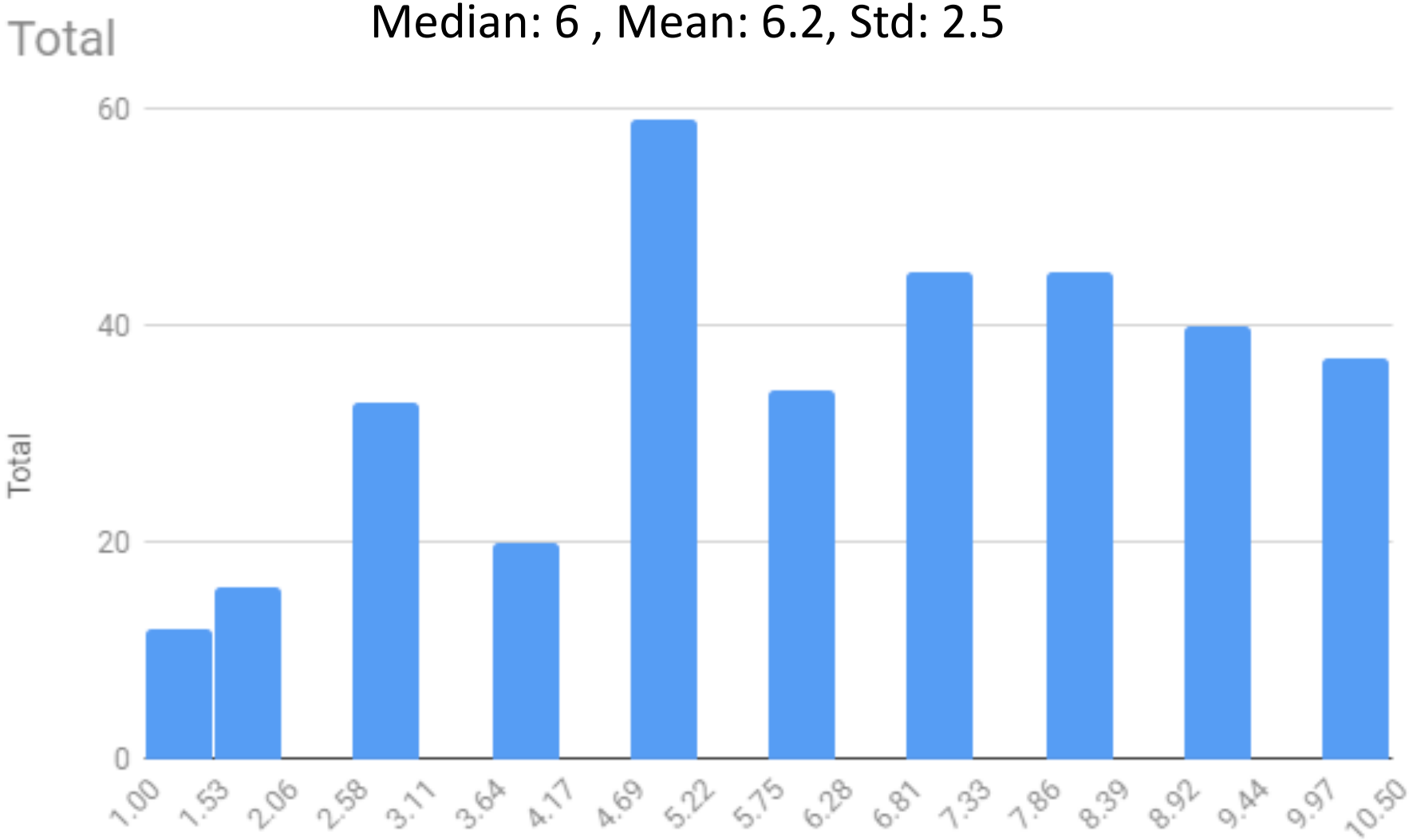
Relevant Reading

- Today
  - Chapters 11 and 14
- Next lecture
  - Chapters 16 and 17



Microquiz 1 Results

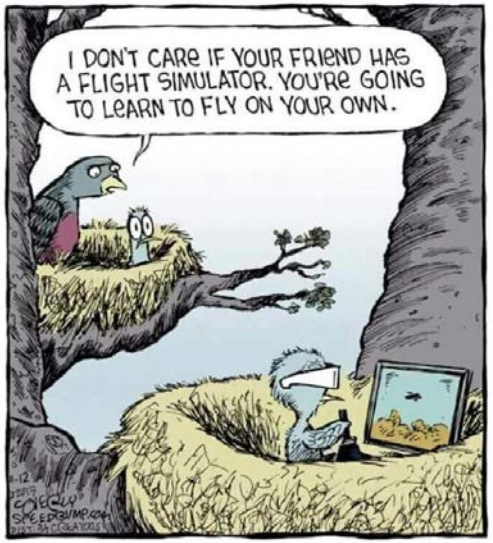
Median: 6 , Mean: 6.2, Std: 2.5



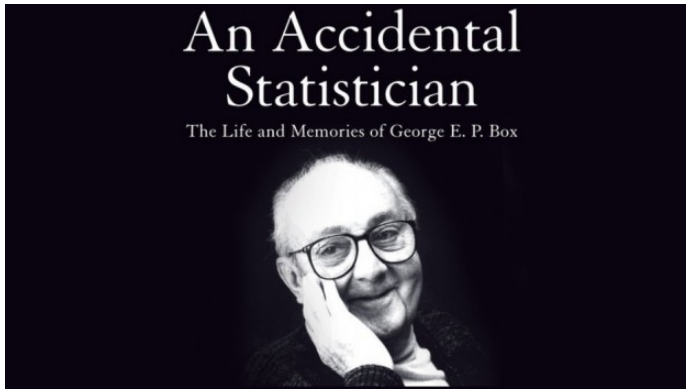
Summary of Last Lecture

- As far as we can tell, the world is stochastic
- Therefore models need to estimate probabilities
- Analytic models feasible for reasonably simple situations
- For more complex situations, simulations often better
- Simulate stochastic process by:
  - Defining an event (e.g., rolling a die N times, flipping a coin N times, randomly picking birthdays for N people),
  - Running some number of trials,
  - Estimating probability of observing particular event (e.g., rolling 11111, finding three people with shared birthday)
-

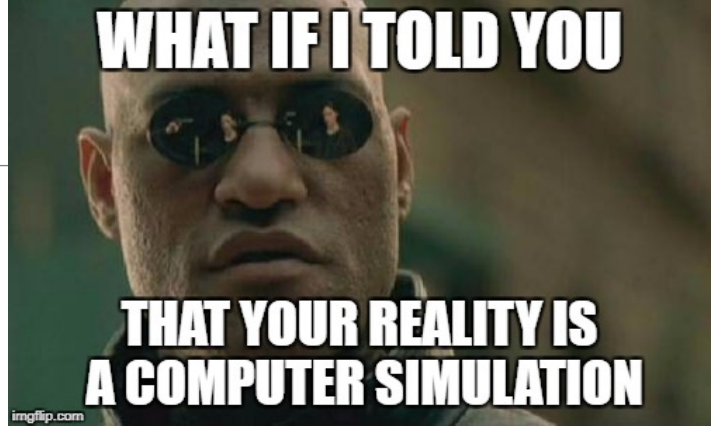
Simulation Model



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

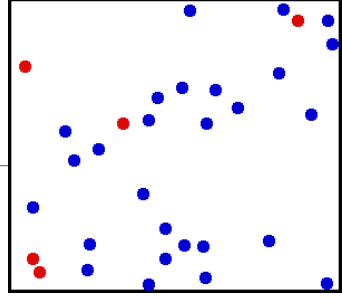


Simulations Are Used a Lot

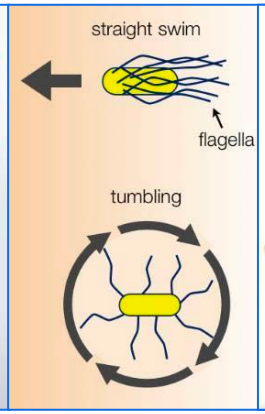


- To model systems that are mathematically intractable
- To extract useful intermediate results
- To support iterative development by successive refinement and asking/answering "what if" questions

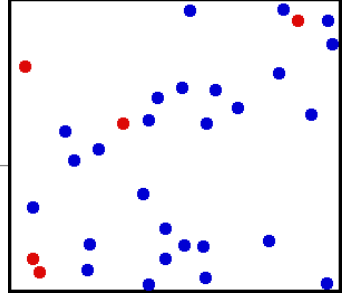
Why Random Walks?



- Important for modeling behavior in many domains
  - Understanding the stock market, modeling diffusion processes, modeling cell movement
- Good illustration of how to use simulations to understand things
- Excuse to cover some important programming topics
  - Practice with classes
  - Practice with plotting



What are Random Walks?



Method to model a system where:

- Objects **wander away** from where they start
- Objects start at a location and **choose a random direction** in which to take each step
  - distribution of choice of direction (and speed) can be different for different object types

Example: Brownian motion of particles of different types

- What are likely physical distributions of different kinds of particles over time?

Random Walks

- Model properties of systems where objects wander at random according to some distribution of steps:

1. Perform an experiment where you take **N steps at random based on some distribution** and ask how far away will you be from the start (or where did you stop)?
2. With many **experiment trials/repetitions**, what is the average distance away from the start location (or what are the properties of the set of end points)?
3. As you increase the number of steps, is there a relation between the number of steps and the average distance away from the start?
4. Is there a description of the set of end points?

Random Walks in the Real World

- In modeling stocks, change in price often modeled as a Gaussian random walk
  - Step size chosen from a normal distribution
  - Basic assumption of Black-Scholes-Merton option pricing model
- In population genetics, random walk describes statistical properties of genetic drift
- In physics, the Brownian motion of molecules in liquids and gases is a random walk

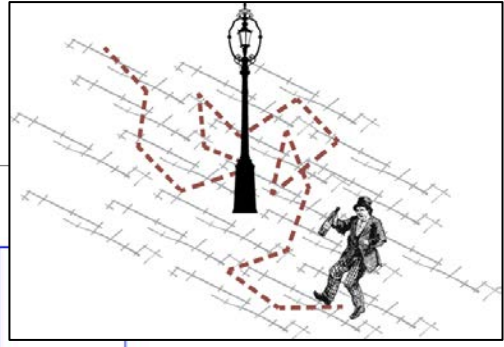
Our Initial Random Walk

- Often called the drunkard's walk
- A random walk on a two-dimensional surface
- Each step a fixed distance
- Step taken in a direction chosen randomly

Odysseus

Homer – Just Odd

Drunkard's Walk



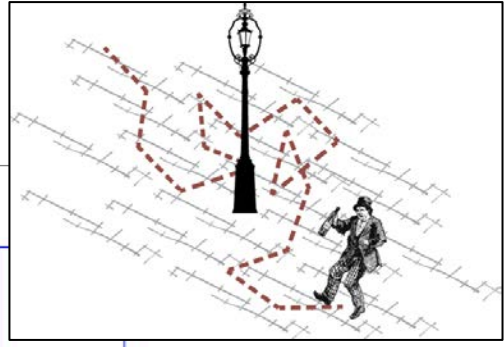
Homer starts at the origin, and takes a unit step in one of the cardinal directions at random with equal likelihood.

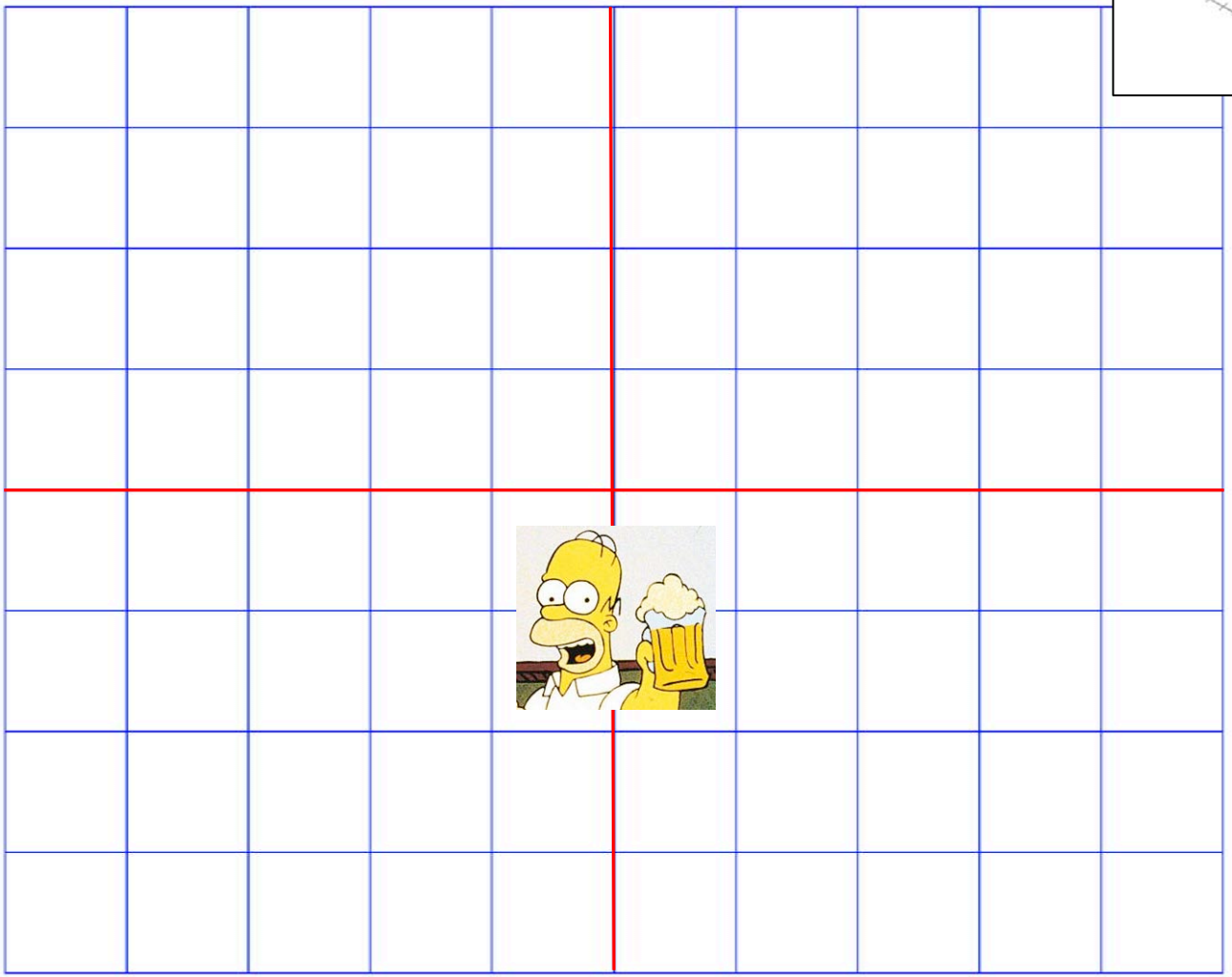
How does distance from origin relate to number of steps?

Poll 1

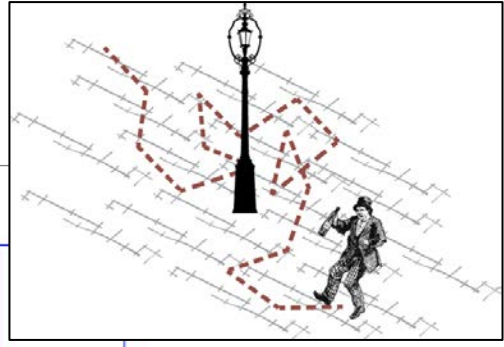
- How far will Homer get?

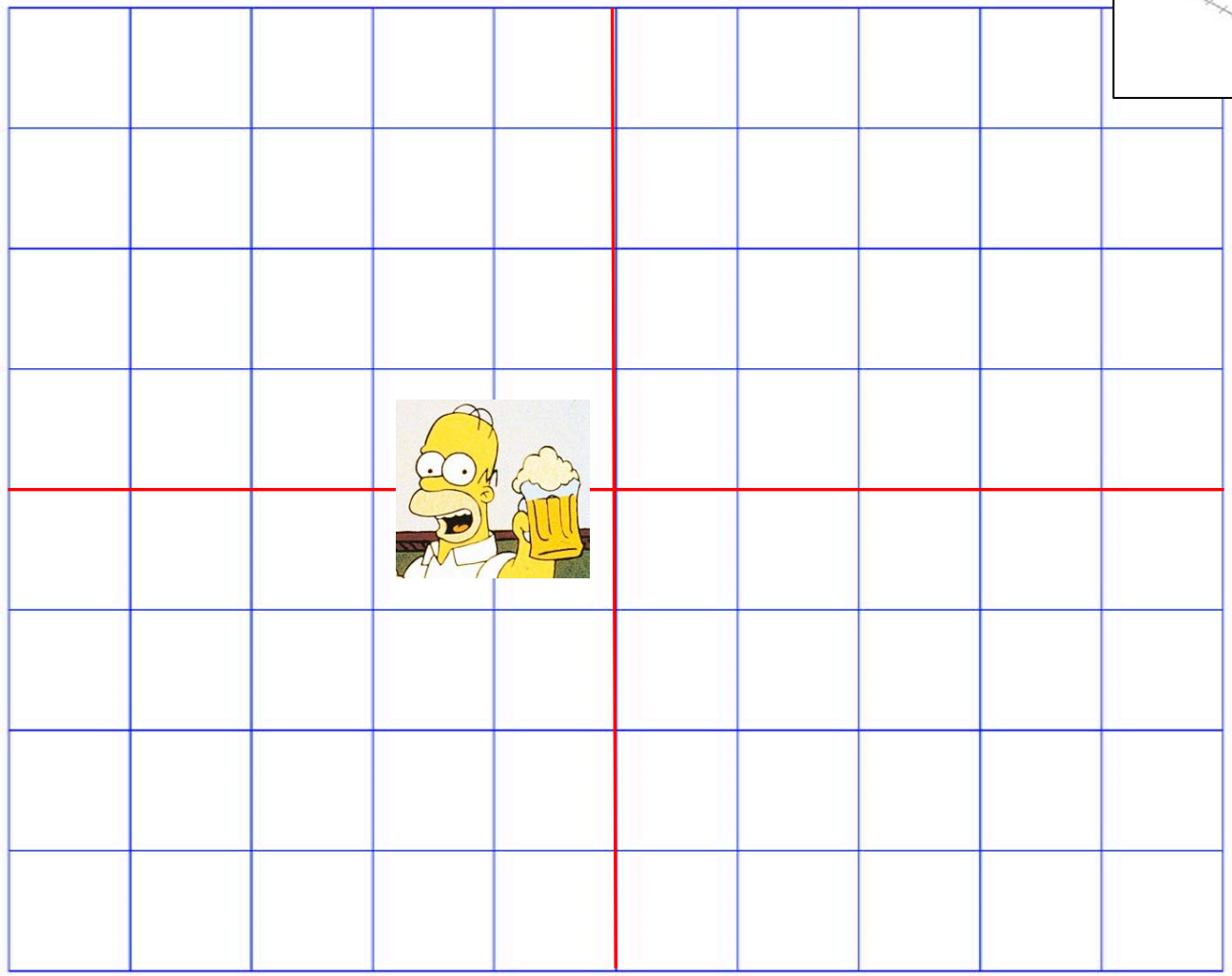
One Possible First Step



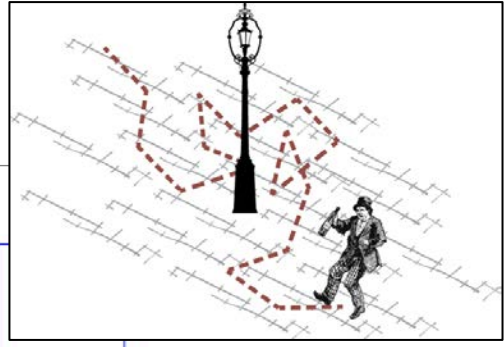


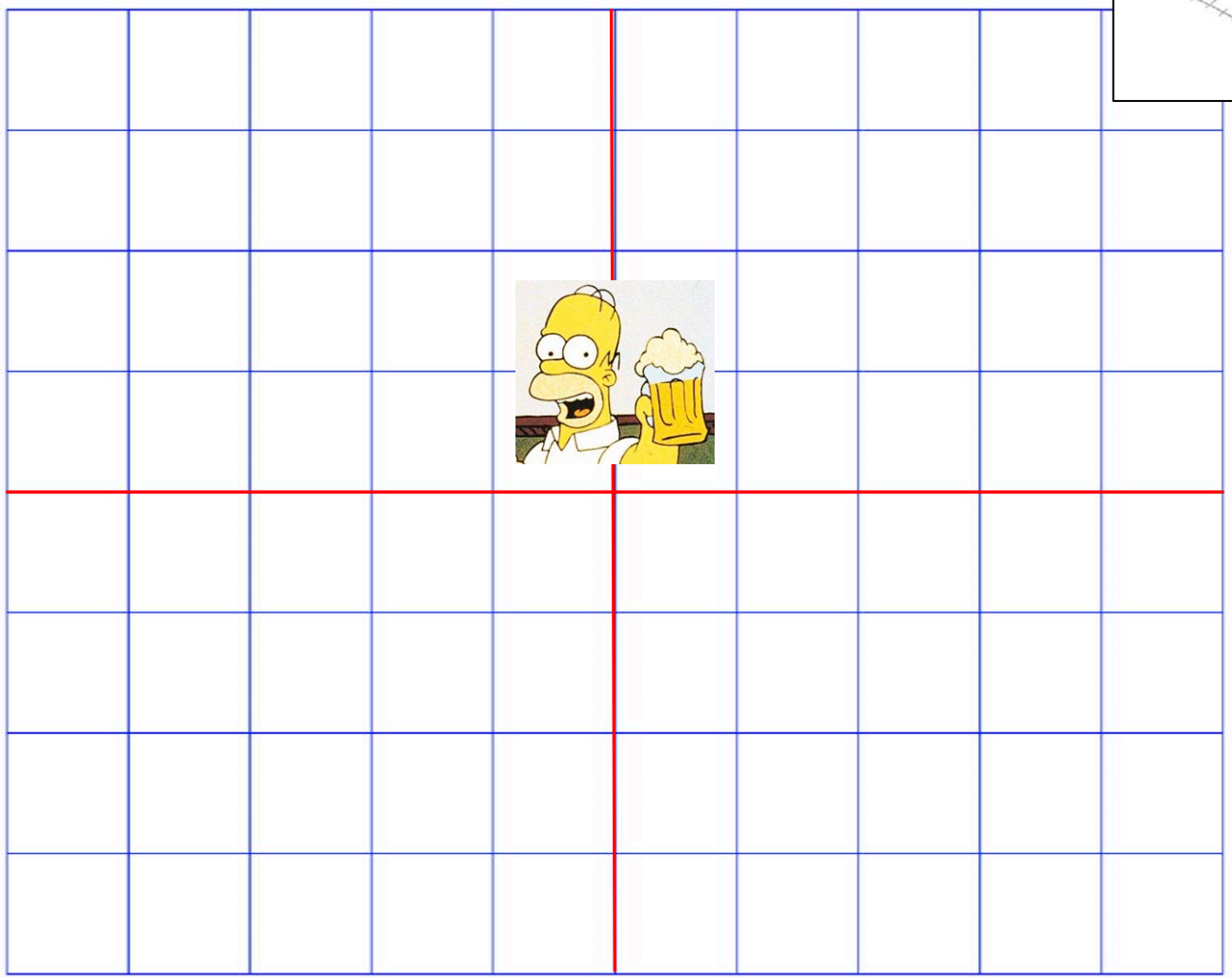
Another Possible First Step



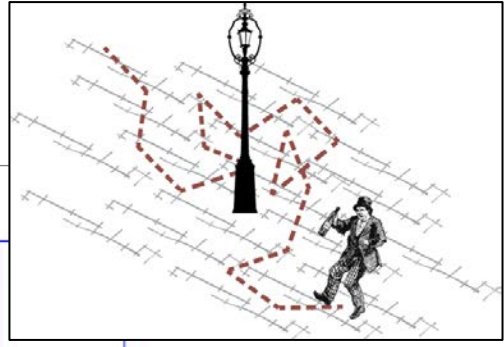


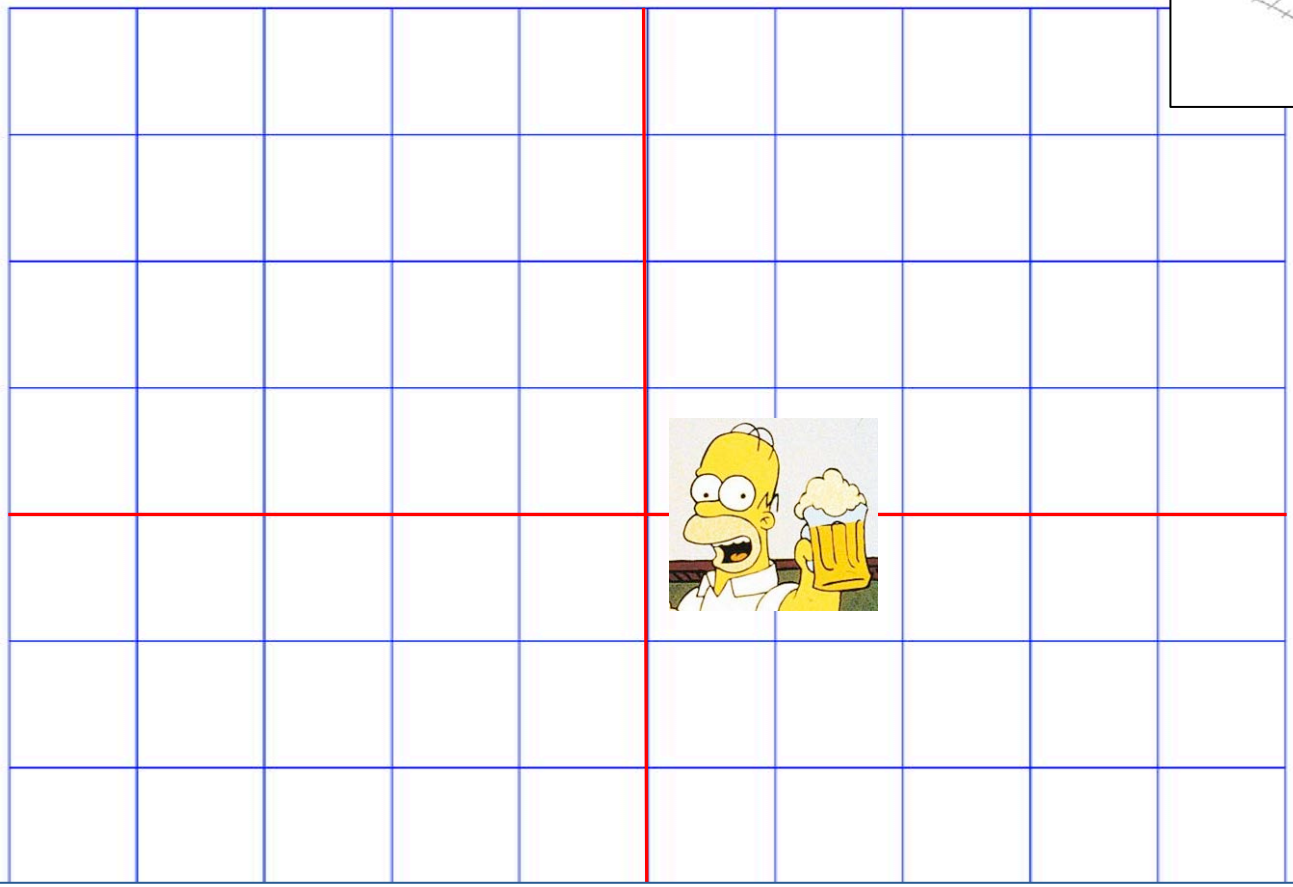
Yet Another Possible First Step





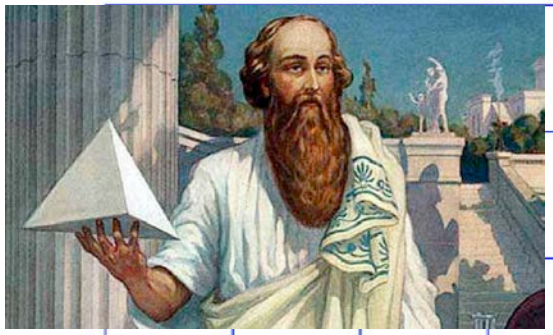
Last Possible First Step



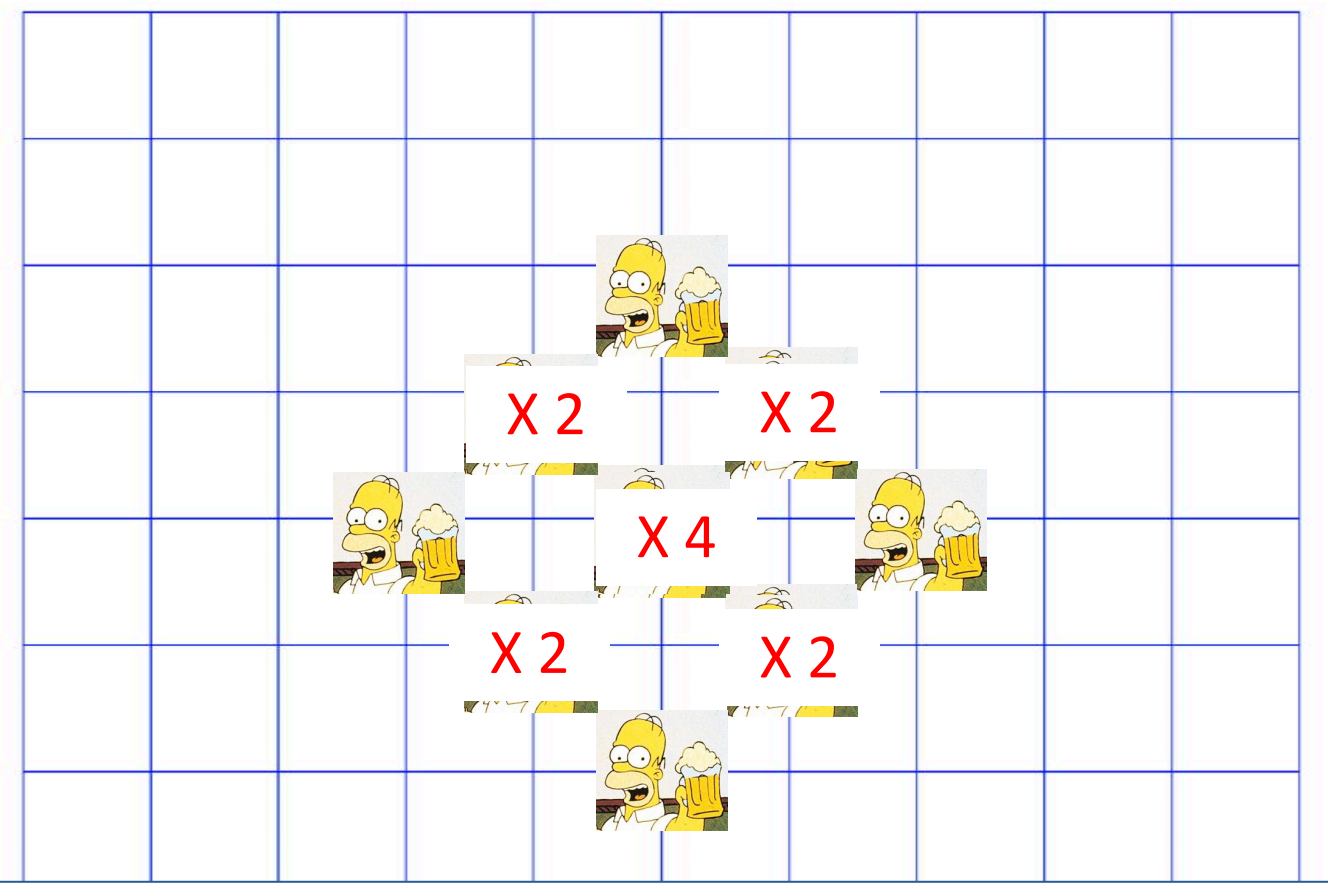


If all steps are equally likely, after one step drunk is distance 1.0 away from start

Possible Distances After Two Steps (assuming moved right on first step, other options similar)



Possible Distances After Two Steps



Average if equally likely is:

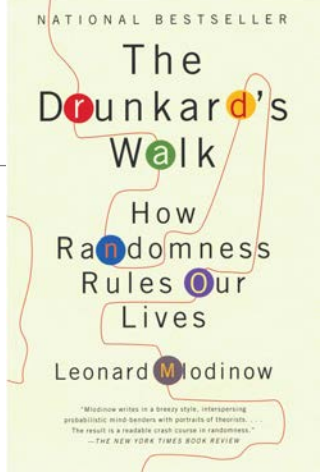
(2 + 2* 1.41 + 2 + 2* 1.41 + 2 + 2* 1.41 + 2 + 2 *1.4 + 4*0)/16

= 1.205

Expected Distance After 100,000 Steps?

- Can see that expected distance seems to increase with number of steps, and certainly maximum distance does
  - Went from **0** to **1** to **1.205** as expected distance
- Enumerating all possibilities (even using a computer) clearly not practical
- Need a different approach to problem
- Will use a stochastic simulation

Structure of Simulation

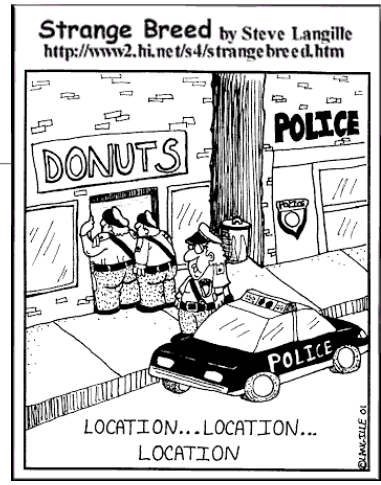


- Simulate one walk of k steps
  - Record distance from origin at end of walk
  - Record final location
- Simulate n such walks, each of k steps
  - Record all distances and final locations
- Report average distance from origin over set of n walks
  - Will come back to final locations over set of walks later

First, Some Useful Abstractions

- Location—a place
  - An object, like a drunk, has a location
  - Location can change as object moves
- Field—a collection of drunks with locations
  - Initially just those locations occupied by drunks
- Drunk—somebody who wanders from location to location in a field

Class Location, part 1




```
class Location(object):
    def __init__(self, x, y):
        """x and y are numbers"""
        self.x = x
        self.y = y

    def move(self, deltaX, deltaY):
        """deltaX and deltaY are numbers"""
        return Location(self.x + deltaX,
                        self.y + deltaY)

    def getX(self):
        return self.x

    def getY(self):
        return self.y
```

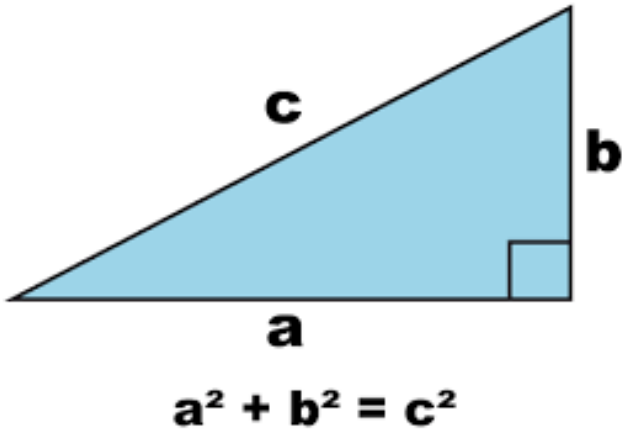
Note that `move` returns a new location

Will let x, y, deltaX and deltaY be floats

More general than example

Class Location, continued

```
def distFrom(self, other):
    xDist = self.x - other.getX()
    yDist = self.y - other.getY()
    return (xDist**2 + yDist**2)**0.5

def __str__(self):
    return '<' + str(self.x) + ', ' + str(self.y) + '>'
```



Euclidean distance

Class Drunk

```
class Drunk(object):
    def __init__(self, name = None):
        """Assumes name is a str"""
        self.name = name

    def __str__(self):
        if self != None:
            return self.name
        return 'Anonymous'
```

Not intended to be useful on its own

A base class to be inherited

The Usual Drunk

```
class UsualDrunk(Drunk):
    def takeStep(self):
        stepChoices = [(0,1), (0,-1),
                       (1, 0), (-1, 0)]
        return random.choice(stepChoices)
```

Takes a random step in cardinal directions with equal probability

Another Subclass of Drunk

```
class MasochistDrunk(Drunk):
    def takeStep(self):
        stepChoices = [(0.0,1.1), (0.0,-0.9),
                       (1.0, 0.0), (-1.0, 0.0)]
        return random.choice(stepChoices)
```

Yet Another Subclass of Drunk

```
class LiberalDrunk(Drunk):
    def takeStep(self):
        stepChoices = [(0.0,1.0), (0.0,-1.0),
                       (0.9, 0.0), (-1.1, 0.0)]
        return random.choice(stepChoices)
```

Poll 2:

Class Field, part 1

```
class Field(object):
    def __init__(self):
        self.drunks = {}

    def addDrunk(self, drunk, loc):
        if drunk in self.drunks:
            raise ValueError('Duplicate drunk')
        else:
            self.drunks[drunk] = loc

    def getLoc(self, drunk):
        if drunk not in self.drunks:
            raise ValueError('Drunk not in field')
        return self.drunks[drunk]
```

Dictionary mapping Drunk to Location; Can have many drunks in a field

Class Field, continued

```
def moveDrunk(self, drunk):
    if drunk not in self.drunks:
        raise ValueError('Drunk not in field')
    xDist, yDist = drunk.takeStep()
    #use move method of Location to get new location
    self.drunks[drunk] =\
        self.drunks[drunk].move(xDist, yDist)
```

Dealing with Location object

So call method to set new location

Remember that move returns a new location by adding deltas to current location

Store that new location in dictionary as entry associated with drunk (as key)

Simulating a Single Walk

```
def walk(f, d, numSteps):
    """Assumes: f a Field, d a Drunk in f,
               and numSteps an int >= 0.
       Moves d numSteps times, and returns the
       distance between the final location and
       the location at the start of the walk."""
    start = f.getLoc(d)
    for s in range(numSteps):
        f.moveDrunk(d)
    return start.distFrom(f.getLoc(d))
```

Move drunk `d` by `numSteps` steps in field `f`

Use method on location of drunk `d` to get distance from `start` location

Simulating Multiple Walks

```
def simWalks(numSteps, numTrials, dClass):
    """Assumes numSteps an int >= 0, numTrials an int > 0,
         dClass a subclass of Drunk
       Simulates numTrials walks of numSteps steps each.
       Returns a list of the final distances for each trial"""
    Homer = dClass('Homer')
    origin = Location(0, 0)
    distances = []
    for t in range(numTrials):
        f = Field()
        f.addDrunk(Homer, origin)
        distances.append(round(walk(f, Homer,
                                    numTrials), 1))
    return distances
```

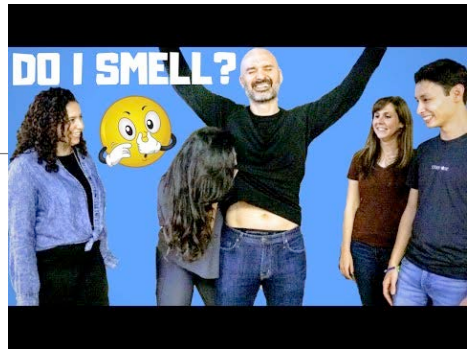
The name of a Python class, not an instance, leads to a generalized function

Poll 3

Putting It All Together

```
def drunkTest(walkLengths, numTrials, dClass):
    """Assumes walkLengths a sequence of ints >= 0
       numTrials an int > 0, dClass a subclass of Drunk
       For each number of steps in walkLengths, runs
       simWalks with numTrials walks and prints results"""
    for numSteps in walkLengths:
        distances = simWalks(numSteps, numTrials, dClass)
        print(dClass.__name__, 'random walk of',
              numSteps, 'steps')
        print(' Mean =',
              round(sum(distances)/len(distances), 4))
        print(' Max =',
              max(distances), 'Min =', min(distances))
```

Let's Try It




```
drunkTest((10, 100, 1000, 10000), 100,
          UsualDrunk)

UsualDrunk random walk of 10 steps
 Mean = 8.634
 Max = 21.6 Min = 1.4
UsualDrunk random walk of 100 steps
 Mean = 8.57
 Max = 22.0 Min = 0.0
UsualDrunk random walk of 1000 steps
 Mean = 9.206
 Max = 21.6 Min = 1.4
UsualDrunk random walk of 10000 steps
 Mean = 8.727
 Max = 23.5 Min = 1.4
```

Should mean distance be same for different numbers of steps?

Should mean distance be almost 9 after 10 steps?

Does this pass the smell test?

Let's Try a Smoke Test

- Try on cases where we think we know the answer
  - A very important precaution!

Sanity Check


```
drunkTest((0, 1, 2) 100, UsualDrunk)

UsualDrunk random walk of 0 steps
 Mean = 8.634
 Max = 21.6 Min = 1.4
UsualDrunk random walk of 1 steps
 Mean = 8.57
 Max = 22.0 Min = 0.0
UsualDrunk random walk of 2 steps
 Mean = 9.206
 Max = 21.6 Min = 1.4
```

Try simulations of 0, 1 and 2 steps

After 0 steps, on average we are 9 units away?

Poll 4

Where's the bug?

Simulating Multiple Walks

```
def simWalks(numSteps, numTrials, dClass):
    """Assumes numSteps an int >= 0, numTrials an int > 0,
         dClass a subclass of Drunk
       Simulates numTrials walks of numSteps steps each.
       Returns a list of the final distances for each trial"""
    Homer = dClass('Homer')
    origin = Location(0, 0)
    distances = []
    for t in range(numTrials):
        f = Field()
        f.addDrunk(Homer, origin)
        distances.append(round(walk(f, Homer,
                                    numSteps ), 1))
    return distances
```

Sanity Check


```
In [18]: drunkTest((0, 1, 2), 100, UsualDrunk)
UsualDrunk random walk of 0 steps
 Mean = 0.0
 Max = 0.0 Min = 0.0
UsualDrunk random walk of 1 steps
 Mean = 1.0
 Max = 1.0 Min = 1.0
UsualDrunk random walk of 2 steps
 Mean = 1.268
 Max = 2.0 Min = 0.0
```

Let's Try It

```
drunkTest((10, 100, 1000, 10000), 100,
          UsualDrunk)

UsualDrunk random walk of 10 steps
 Mean = 2.863
 Max = 7.2 Min = 0.0
UsualDrunk random walk of 100 steps
 Mean = 8.296
 Max = 21.6 Min = 1.4
UsualDrunk random walk of 1000 steps
 Mean = 27.297
 Max = 66.3 Min = 4.2
UsualDrunk random walk of 10000 steps
 Mean = 89.241
 Max = 226.5 Min = 10.0
```

And the Masochistic Drunk?

```
random.seed(0)
simAll((UsualDrunk, MasochistDrunk),
       (1000, 10000), 100)
```

```
UsualDrunk random walk of 1000 steps
 Mean = 26.828
 Max = 66.3 Min = 4.2
UsualDrunk random walk of 10000 steps
 Mean = 90.073
 Max = 210.6 Min = 7.2
```

Average step is (0.0, 0.0)

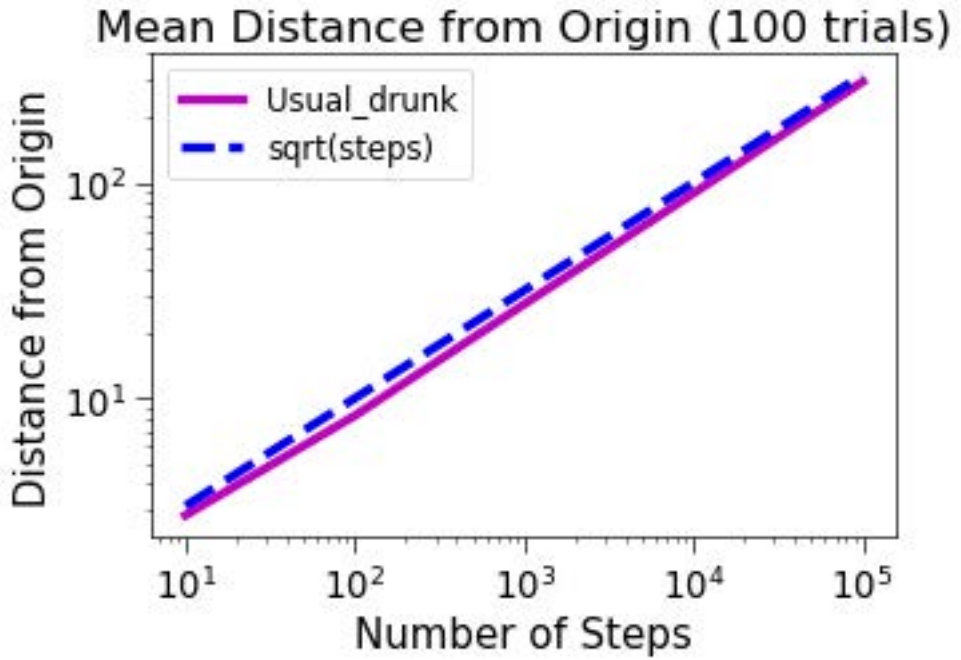
```
MasochistDrunk random walk of 1000 steps
 Mean = 58.425
 Max = 133.3 Min = 6.7
MasochistDrunk random walk of 10000 steps
 Mean = 515.575
 Max = 694.6 Min = 377.7
```

Average step is (0.0, 0.05)

Visualizing the Trend

- Simulate walks of multiple lengths for UsualDrunk
- Seeing distances as numbers helps
- But, better to **plot** distance at end of each length walk for each kind of drunk

Sheds light on why molecular diffusion leads to slow mixing



Plotting (Chapter 11)

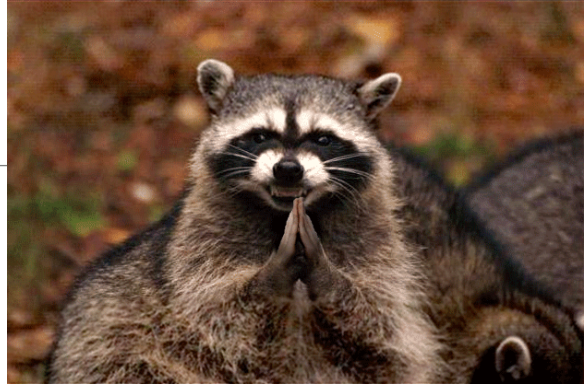
SLIDES DISTRIBUTED, BUT WILL NOT GO THROUGH THEM IN LECTURE

CODE ON SLIDES IN plotting.py

Poll (Anonymous, Idle Curiosity)

Five Minute Break

WHY PLOTTING?



- Sooner or later, everyone needs to produce plots
  - As we look at data in 6.0002 half of term, we will make extensive use of plots to visualize results
- Example of **leveraging an existing library**, rather than writing procedures from scratch
  - See problem sets
- Python provides libraries for (among other topics):
  - Plotting
  - Numerical computation
  - Stochastic computation
  - Machine learning

Very similar to Matlab

matplotlib

- **import library** into computing environment

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

  - Allows **code to reference library** procedures 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 `matplotlib`

- Will see many other examples and details of these ideas in rest of term

A SIMPLE EXAMPLE

```
nVals = []
linear = []
quadratic = []
cubic = []
exponential = []

for n in range(0, 30):
    nVals.append(n)
    linear.append(n)
    quadratic.append(n**2)
    cubic.append(n**3)
    exponential.append(1.5**n)
```

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

PLOTTING THE DATA

- To generate a plot:

X values

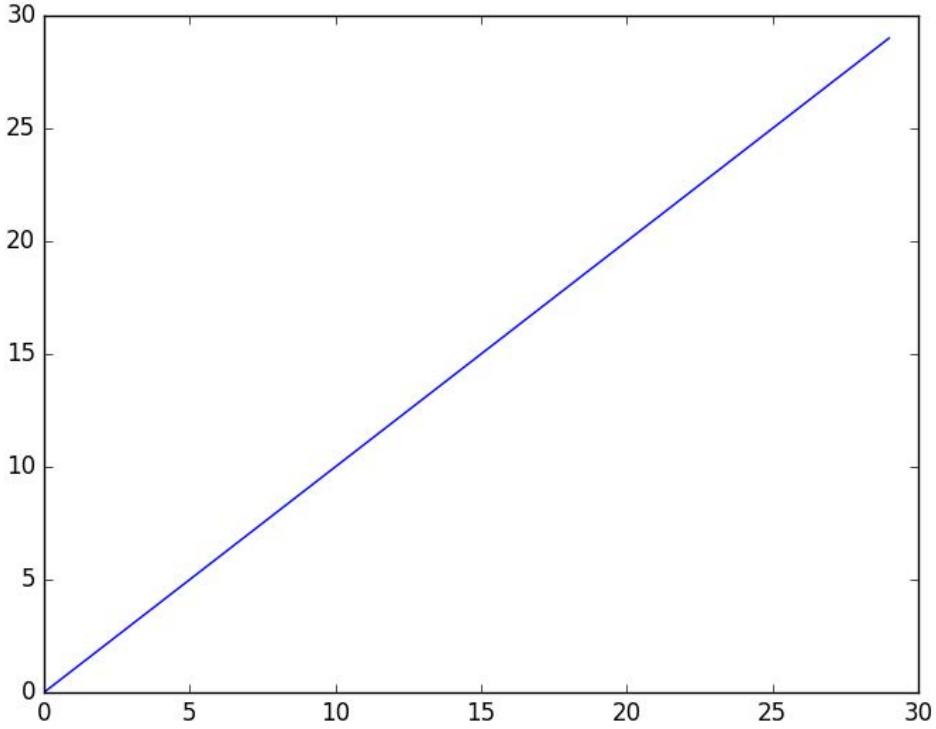
Y values

```
plt.plot(n, <function of n>)
```

- Arguments are lists of numbers (could also be arrays)
  - Must be of the same length
  - Generates a sequence of <x, y> values
  - Plotted in order, then connected with lines
- Calling function in an **iPython** console will generate plots within that console
- Calling function in a **Python** console will create a separate window in which plot is displayed

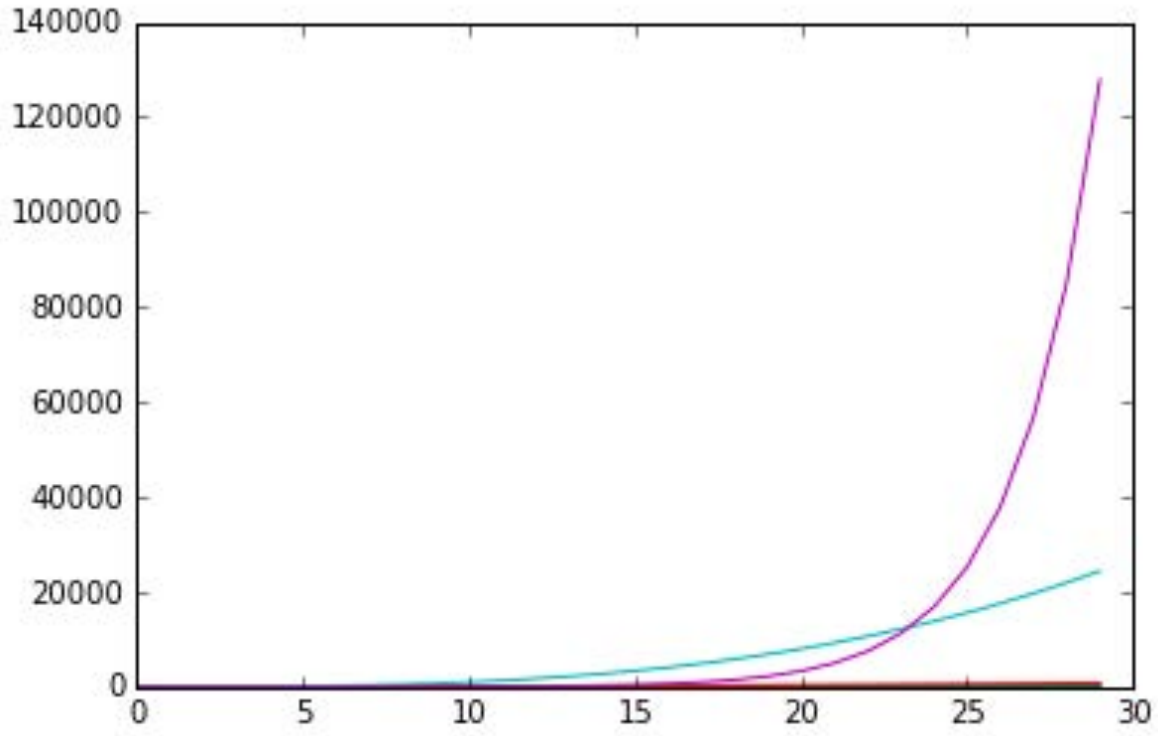
EXAMPLE

```
plt.plot(nVals, linear)
```



SHOWING ALL DATA ON ONE PLOT

```
plt.plot(nVals, linear)
plt.plot(nVals, quadratic)
plt.plot(nVals, cubic)
plt.plot(nVals, exponential)
```



Impossible to see linear graph, or even quadratic graph

Problem is that scales are so different

PRODUCING MULTIPLE PLOTS

- Let's graph each one separately
- Call

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

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

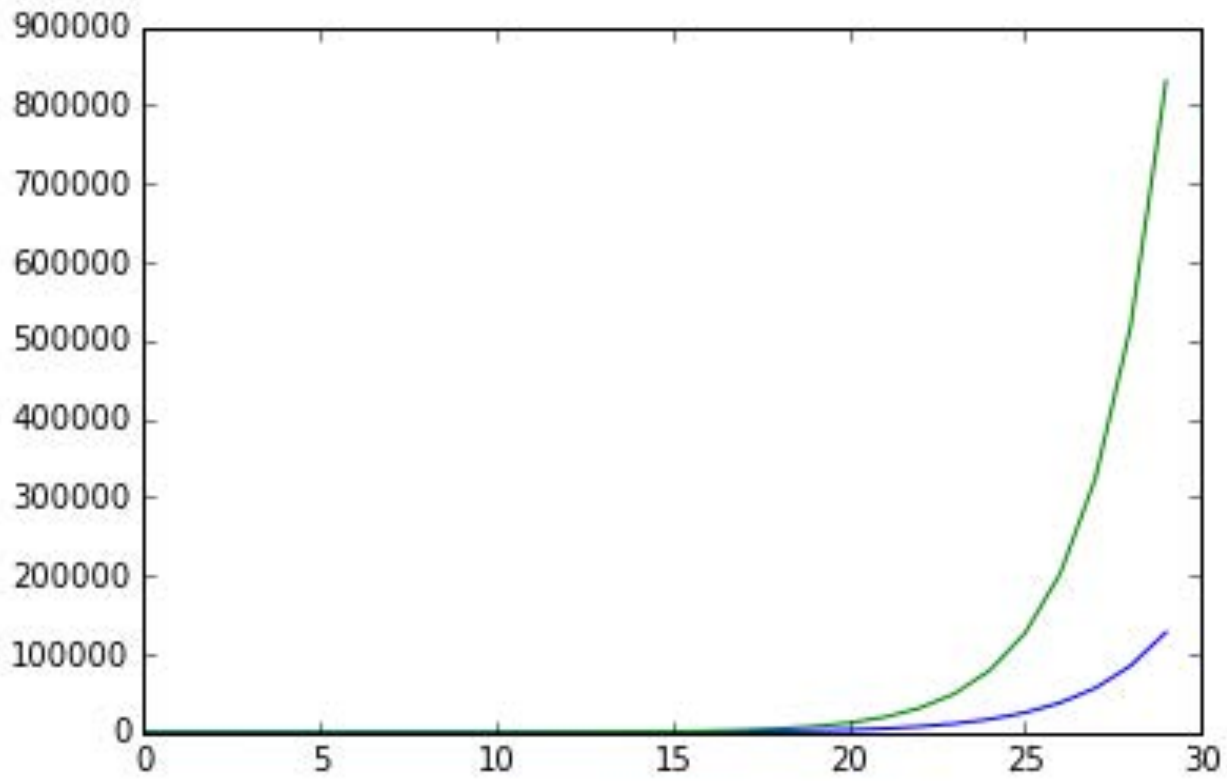
  - Creates a new display with that name if one does not already exist
  - If a display with that name exists, reopens it for processing

EXAMPLE CODE

```
plt.figure('expo')
plt.plot(nVals, exponential)
plt.figure('lin')
plt.plot(nVals, linear)
plt.figure('quad')
plt.plot(nVals, quadratic)
plt.figure('cube')
plt.plot(nVals, cubic)
newExpo = []
for i in range(30):
    newExpo.append(1.6**i)
plt.figure('expo')
plt.plot(nVals, newExpo)
```

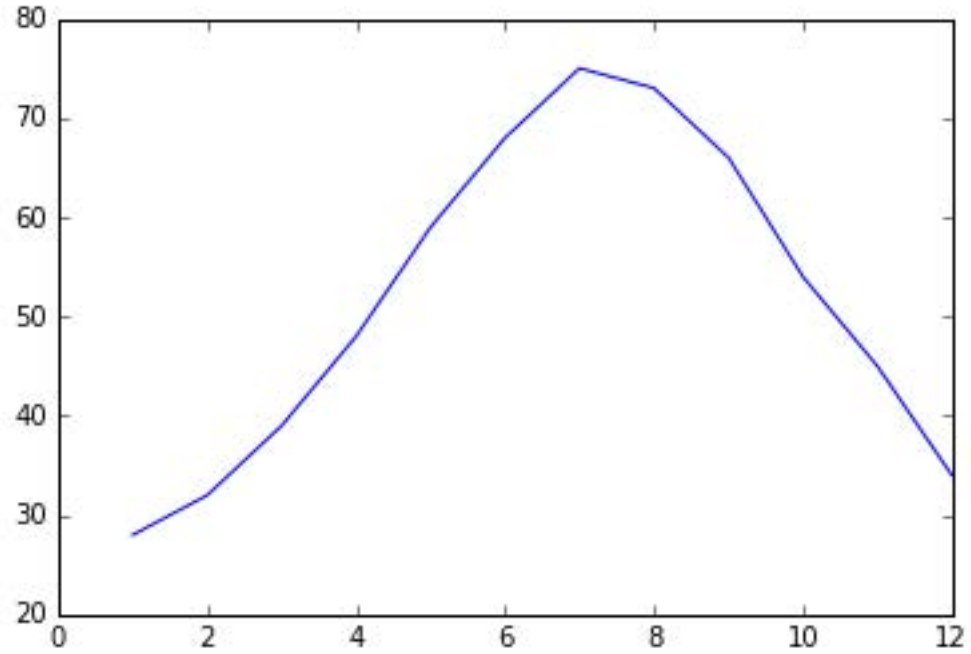
Go back to expo

PLOT WITH EXPONENTIALS



A "REAL" EXAMPLE

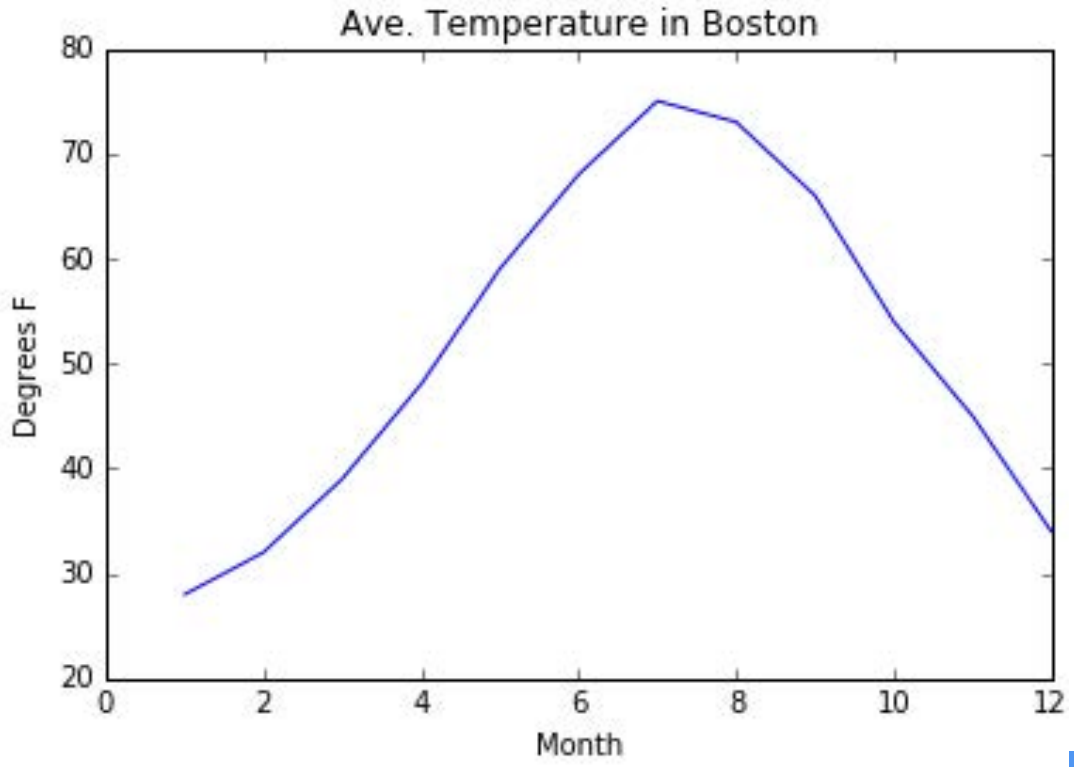
```
months = range(1, 13, 1)
temps = [28,32,39,48,59,68,75,73,66,54,45,34]
plt.plot(months, temps)
```



But what is this trying to tell us?

A "REAL" EXAMPLE

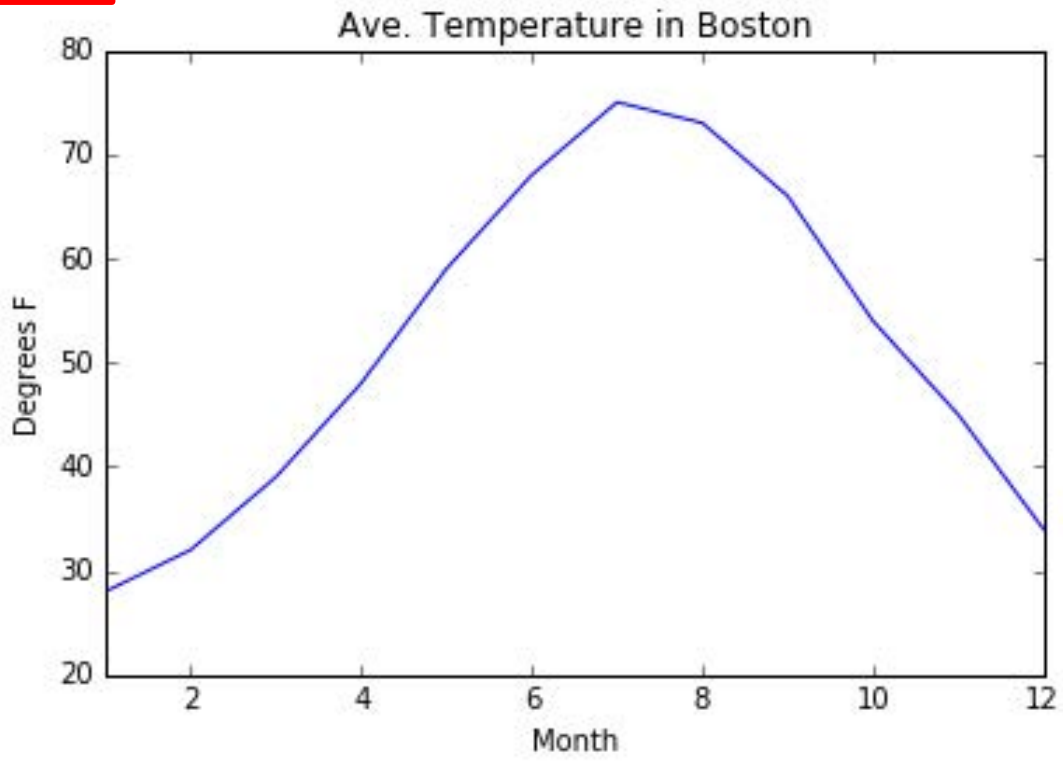
```
months = range(1, 13, 1)
temps = [28,32,39,48,59,68,75,73,66,54,45,34]
plt.plot(months, temps)

plt.title('Ave. Temperature in Boston')
plt.xlabel('Month')
plt.ylabel(('Degrees F'))
```



Still a bit weird looking

A "REAL" EXAMPLE

```
months = range(1, 13, 1)
temps = [28,32,39,48,59,68,75,73,66,54,45,34]
plt.plot(months, temps)

plt.title('Ave. Temperature in Boston')
plt.xlabel('Month')
plt.ylabel(('Degrees F'))
plt.xlim(1, 12)
```

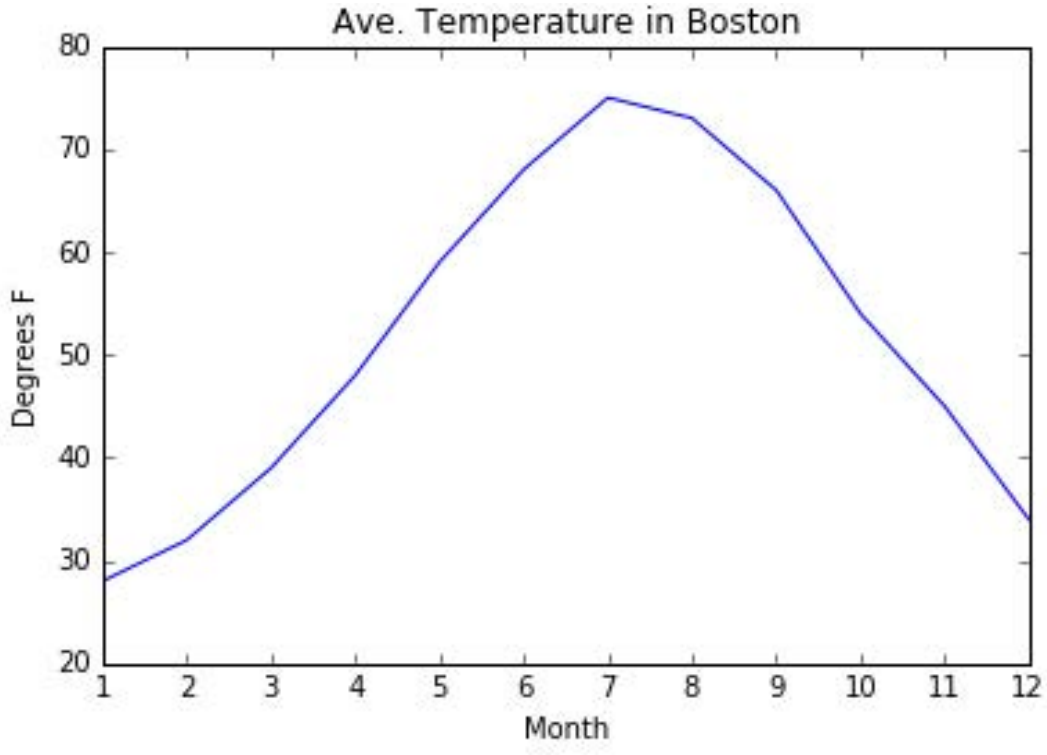


Suppose I want to see each month on x-axis?

A "REAL" EXAMPLE

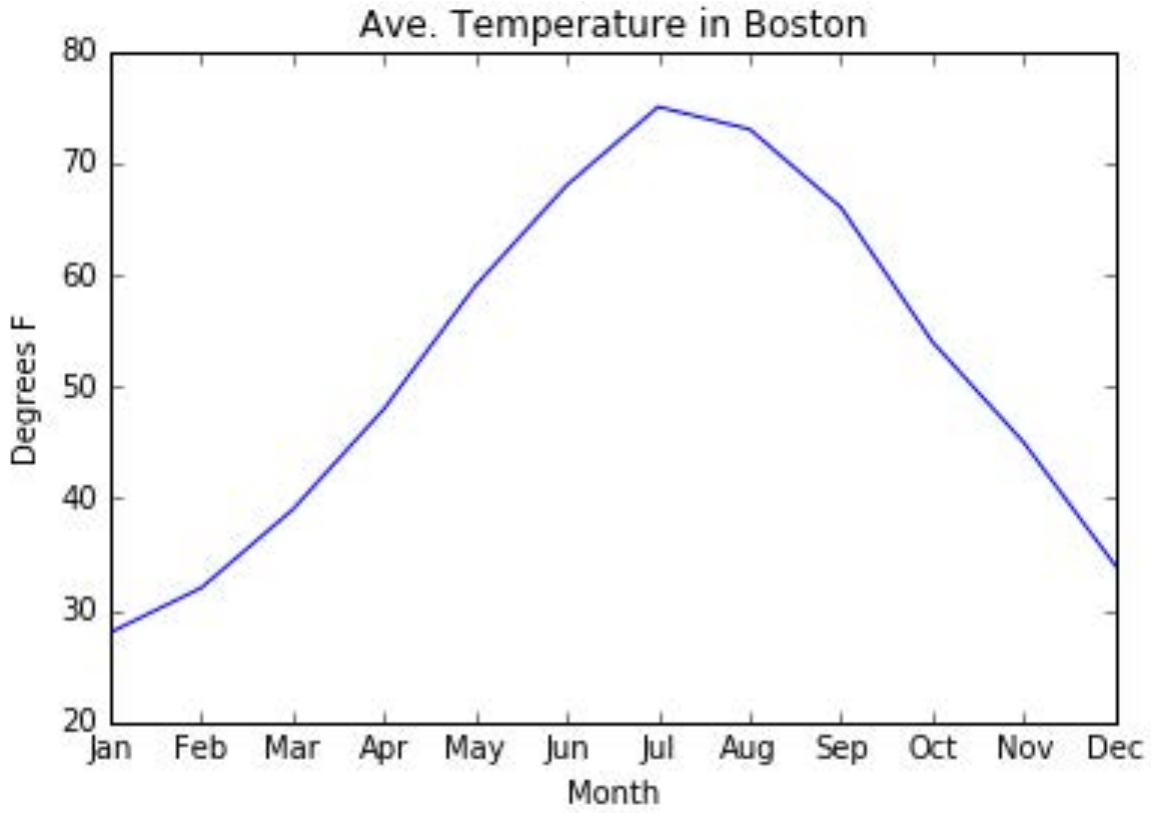
```
months = range(1, 13, 1)
temps = [28,32,39,48,59,68,75,73,66,54,45,34]
plt.plot(months, temps)

plt.title('Ave. Temperature in Boston')
plt.xlabel('Month')
plt.ylabel(('Degrees F'))
plt.xticks((1,2,3,4,5,6,7,8,9,10,11,12))
```

But what about those who can't map numbers to months?



A "REAL" EXAMPLE

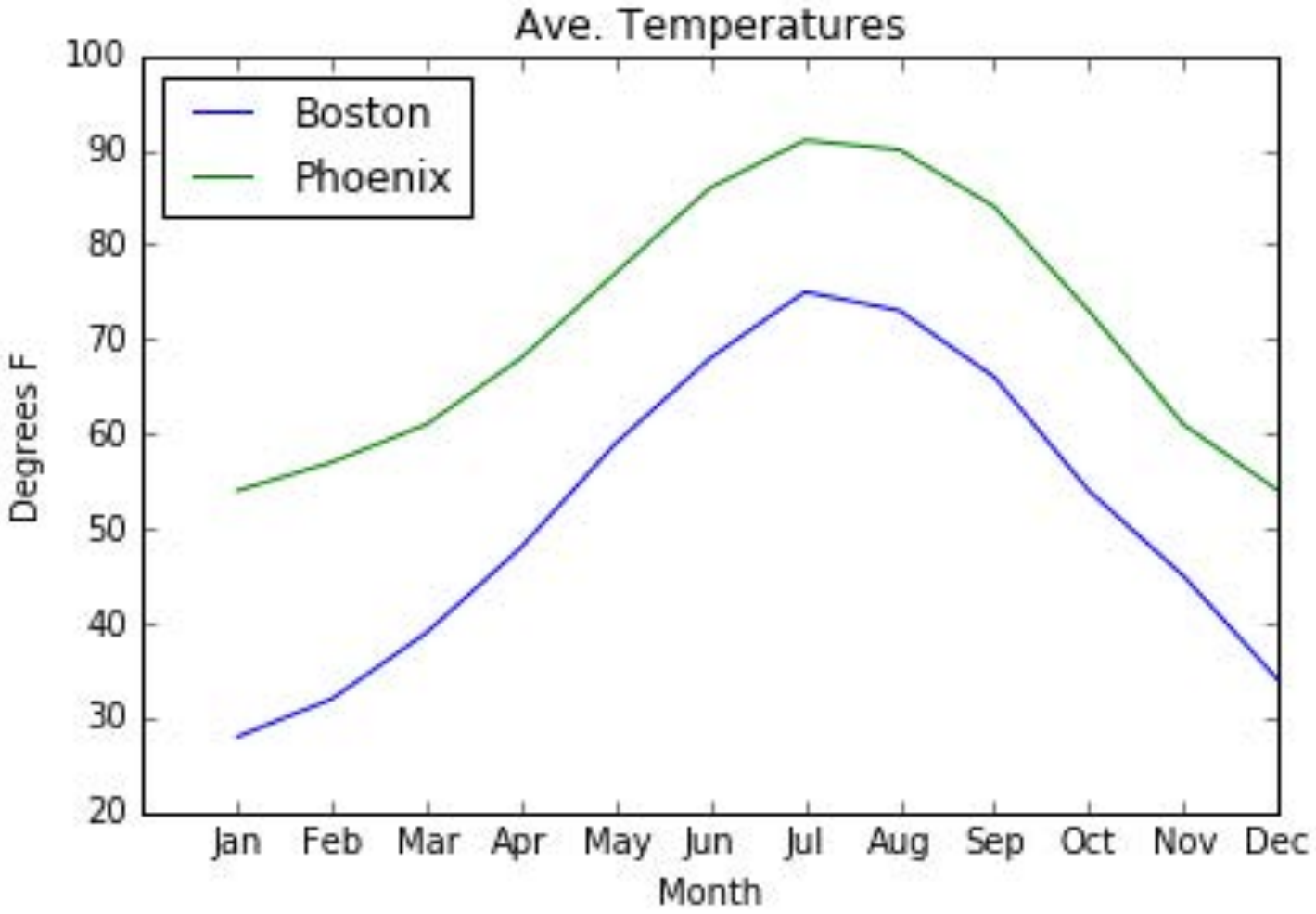
```
plt.xticks((1,2,3,4,5,6,7,8,9,10,11,12),
           ('Jan','Feb','Mar','Apr','May','Jun',
            'Jul','Aug','Sep','Oct','Nov','Dec'))
```



LET'S ADD ANOTHER CITY

```
months = range(1, 13, 1)
boston = [28,32,39,48,59,68,75,73,66,54,45,34]
plt.plot(months, boston, label = 'Boston')
phoenix = [54,57,61,68,77,86,91,90,84,73,61,54]
plt.plot(months, phoenix, label = 'Phoenix')
plt.legend(loc = 'best')

plt.title('Ave. Temperatures')
plt.xlabel('Month')
plt.ylabel(('Degrees F'))
```

PLOT WITH TWO CURVES



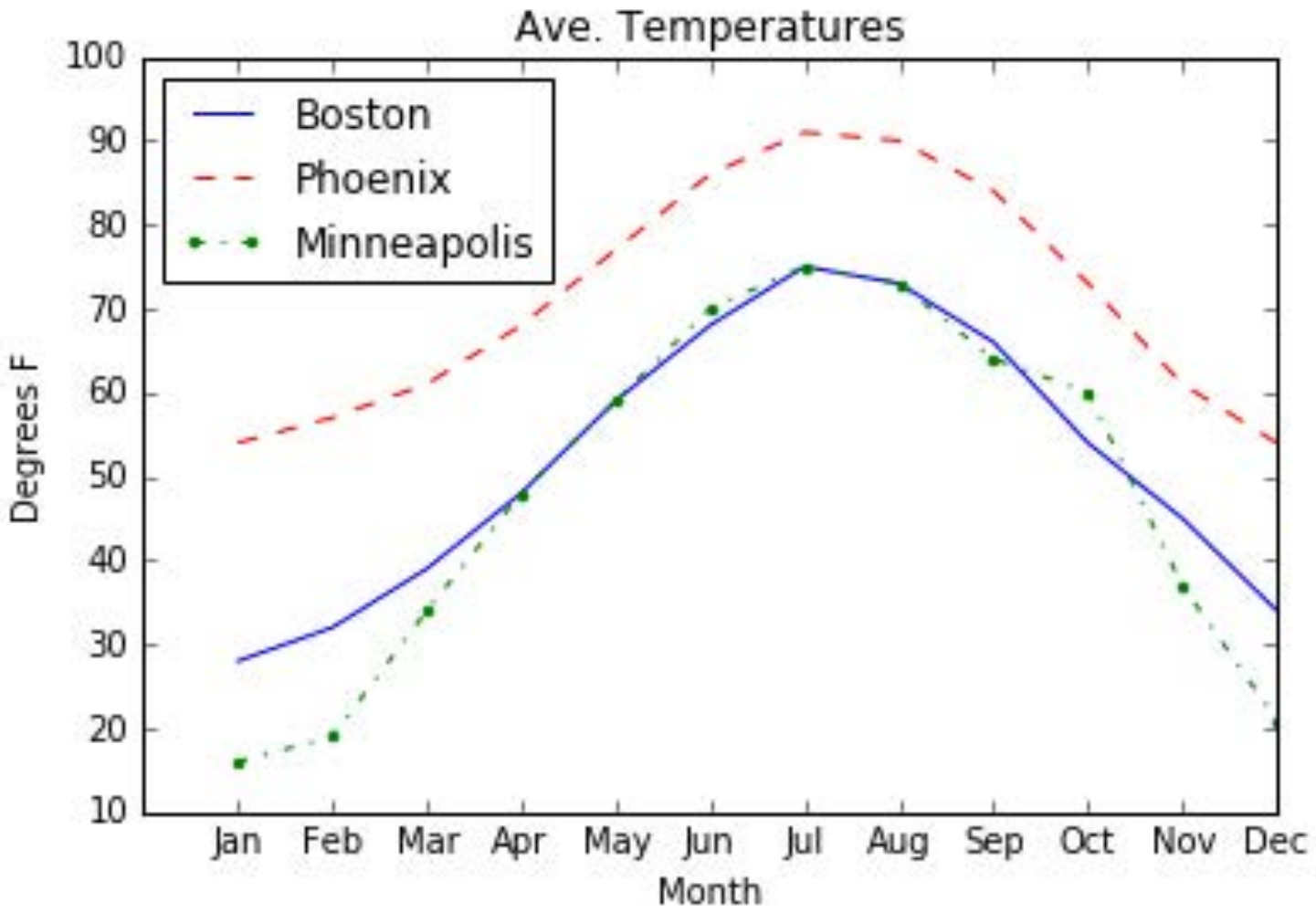
CONTROLLING DISPLAY PARAMETERS

- Suppose we want to control details of the displays themselves
- Examples:
  - changing color or style of data sets
  - changing width of lines or displays

CONTROLLING COLOR AND STYLE

```
months = range(1, 13, 1)
boston = [28,32,39,48,59,68,75,73,66,54,45,34]
plt.plot(months, boston, 'b-', label = 'Boston')
phoenix = [54,57,61,68,77,86,91,90,84,73,61,54]
plt.plot(months, phoenix, 'r--', label = 'Phoenix')
msp = [16,19,34,48,59,70,75,73,64,60,37,21]
plt.plot(months, msp, 'g.-.', label = 'Minneapolis')
plt.legend(loc = 'best')
```

CONTROLLING COLOR AND STYLE



OTHER PLOTTING OPTIONS

- There are many other options for controlling plotting of data
  - `semilogy, semilogx` allow you to use logarithmic scaling on either axis
  - Choice of where to place legend box
  - Choice of "font" for a graph
  - Multiple subplots within a larger display
- Check out textbook and documentation for pyplot to learn more

BACK TO OUR WANDERING DRUNK

Let's add one more type of drunk

```
class LiberalMasochistDrunk(MasochistDrunk):
    def takeStep(self):
        if random.choice([True, False]):
            stepChoices = [(0.0,1.0), (0.0,-1.0),
                           (0.9, 0.0), (-1.1, 0.0)]
            return random.choice(stepChoices)
        else:
            return MasochistDrunk.takeStep(self)
```

Tends to want to go north, but also tends to want to go "left"; does this with equal probability

Simulation with Visualization

```
def simAll(drunkKinds, walkLengths, numTrials):
    styleChoice = styleIterator(('m-', 'b--', 'g-.', 'k'))
    for dClass in drunkKinds:
        curStyle = styleChoice.nextStyle()
        print('Starting simulation of', dClass.__name__)
        means = simDrunk(numTrials, dClass, walkLengths)
        plt.plot(walkLengths, means, curStyle,
                 label = dClass.__name__)
    plt.title('Mean Distance from Origin ('
              + str(numTrials) + ' trials)')
    plt.xlabel('Number of Steps')
    plt.ylabel('Distance from Origin')
    plt.legend(loc = 'best')
```

Some Details

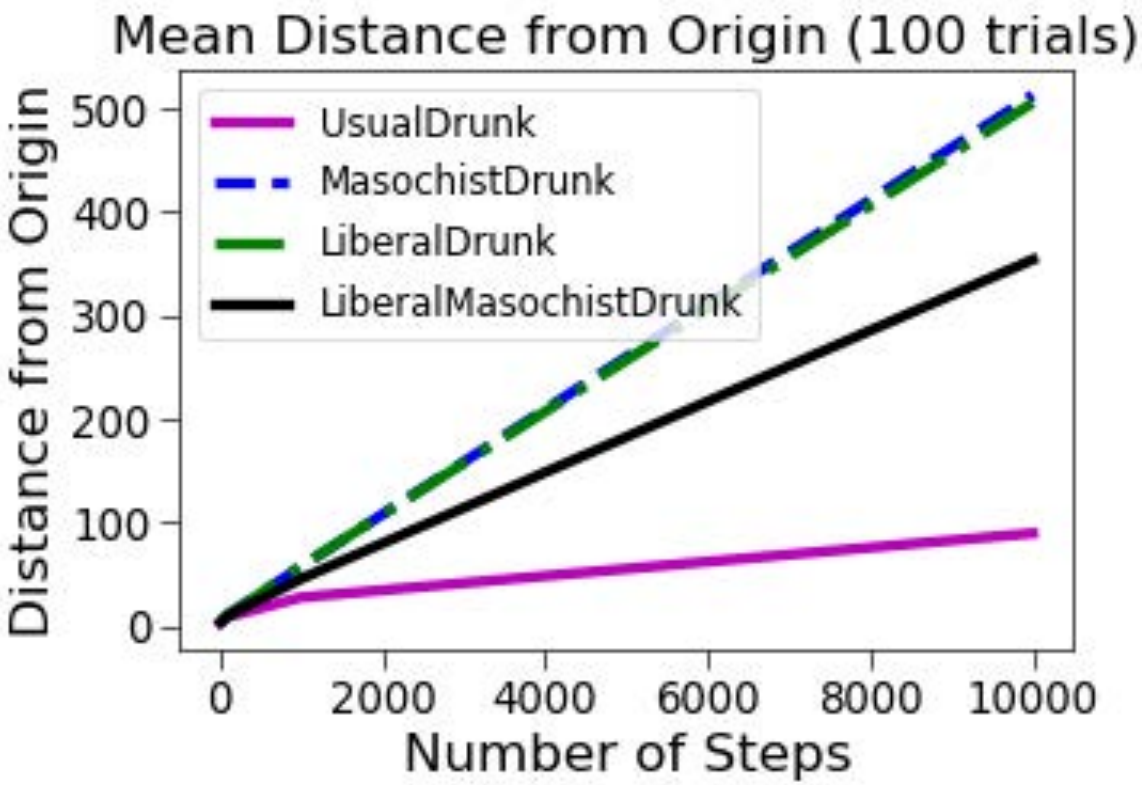
```
class styleIterator(object):
    def __init__(self, styles):
        self.index = 0
        self.styles = styles

    def nextStyle(self):
        result = self.styles[self.index]
        if self.index == len(self.styles) - 1:
            self.index = 0
        else:
            self.index += 1
        return result
```

simDrunk

```
def simDrunk(numTrials, dClass, walkLengths):
    meanDistances = []
    for numSteps in walkLengths:
        print('Starting simulation of',
              numSteps, 'steps')
        trials = simWalks(numSteps, numTrials, dClass)
        mean = sum(trials)/len(trials)
        meanDistances.append(mean)
    return meanDistances
```

Poll 5

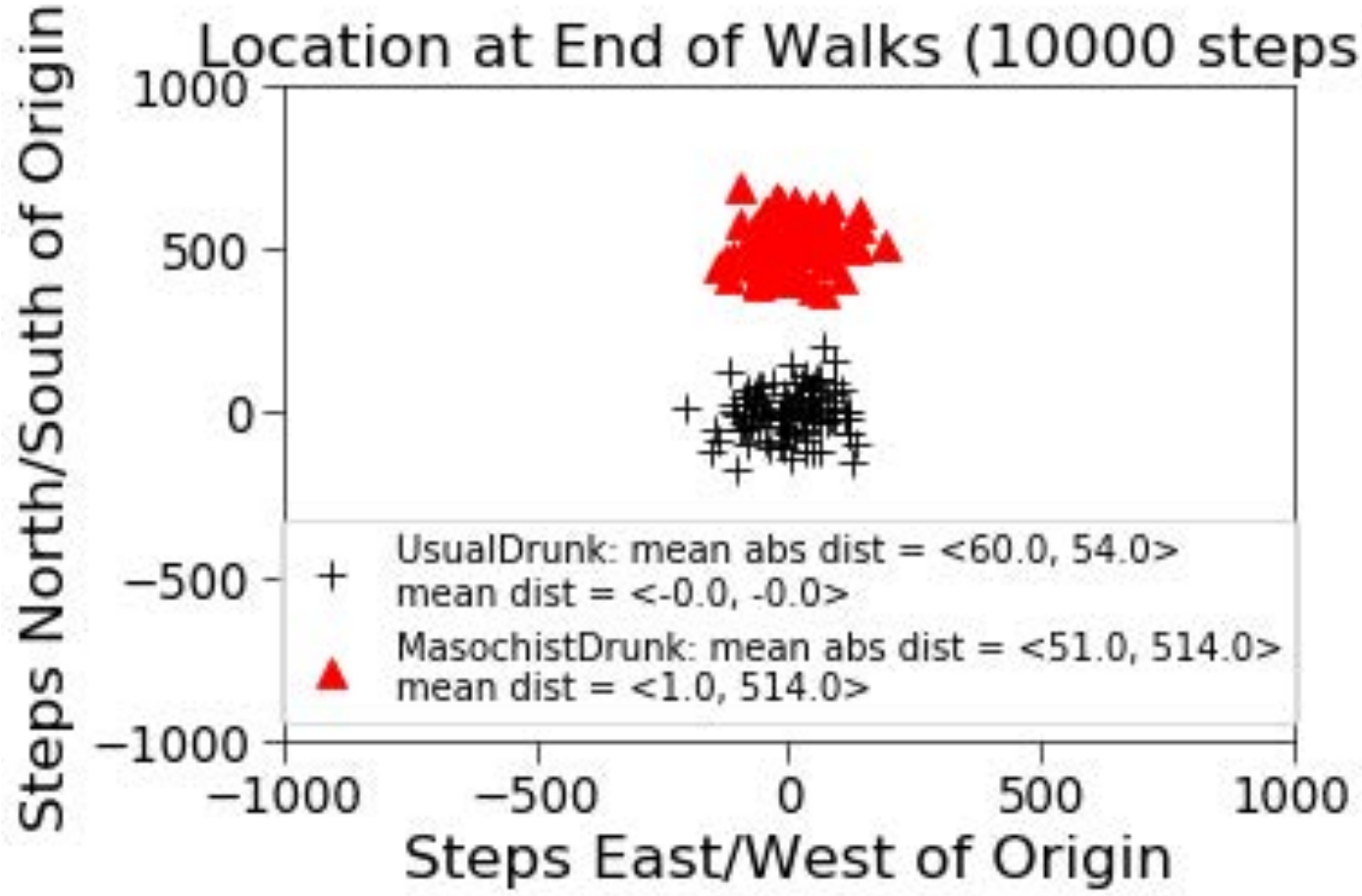
Distance Trends



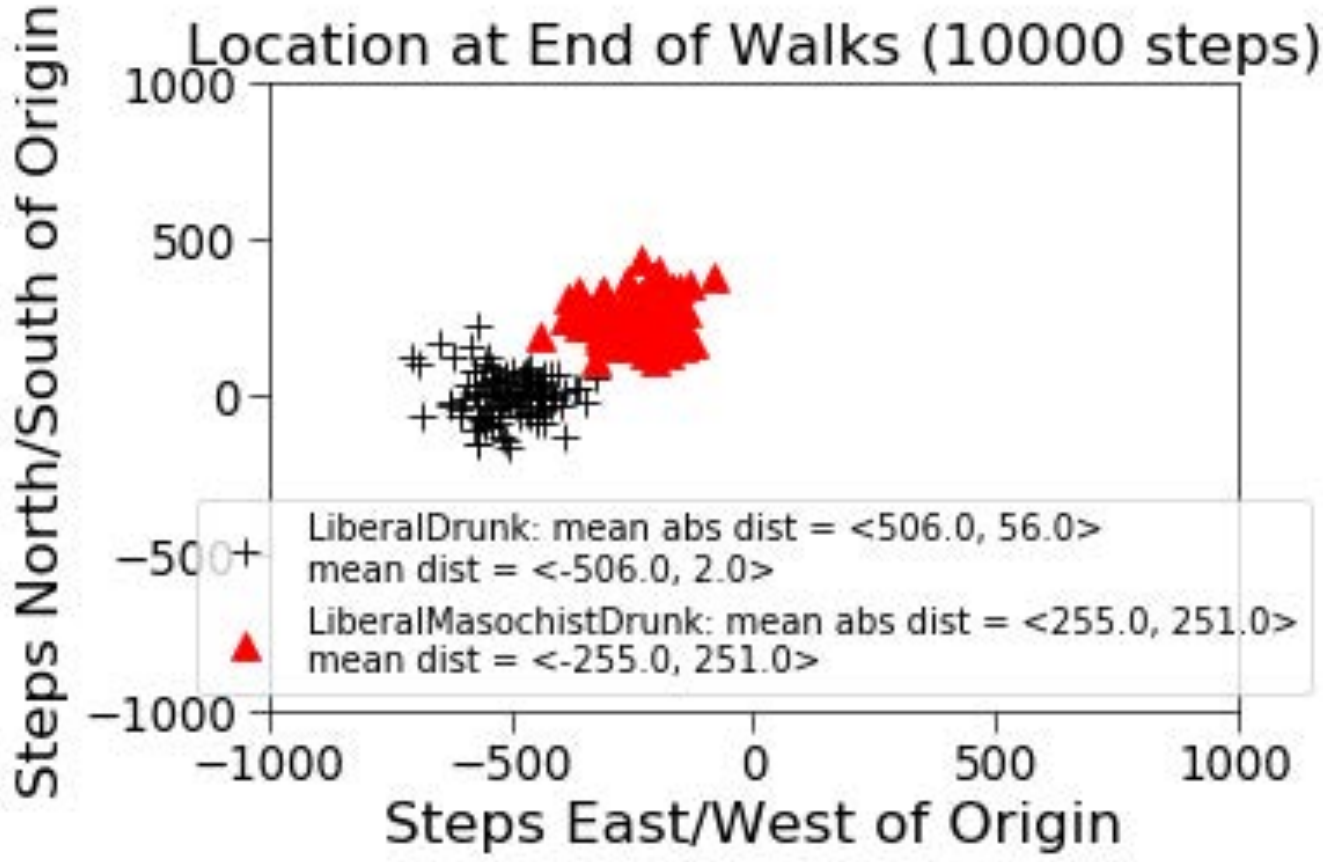
Ending Locations

- Run same simulations as before
  - Consider different kinds of drunks
  - Simulate random walks of different lengths
  - Keep track of final locations
- Plot distribution of final locations
- Report mean absolute distance from origin, and mean actual distance from origin
- See code handout for details

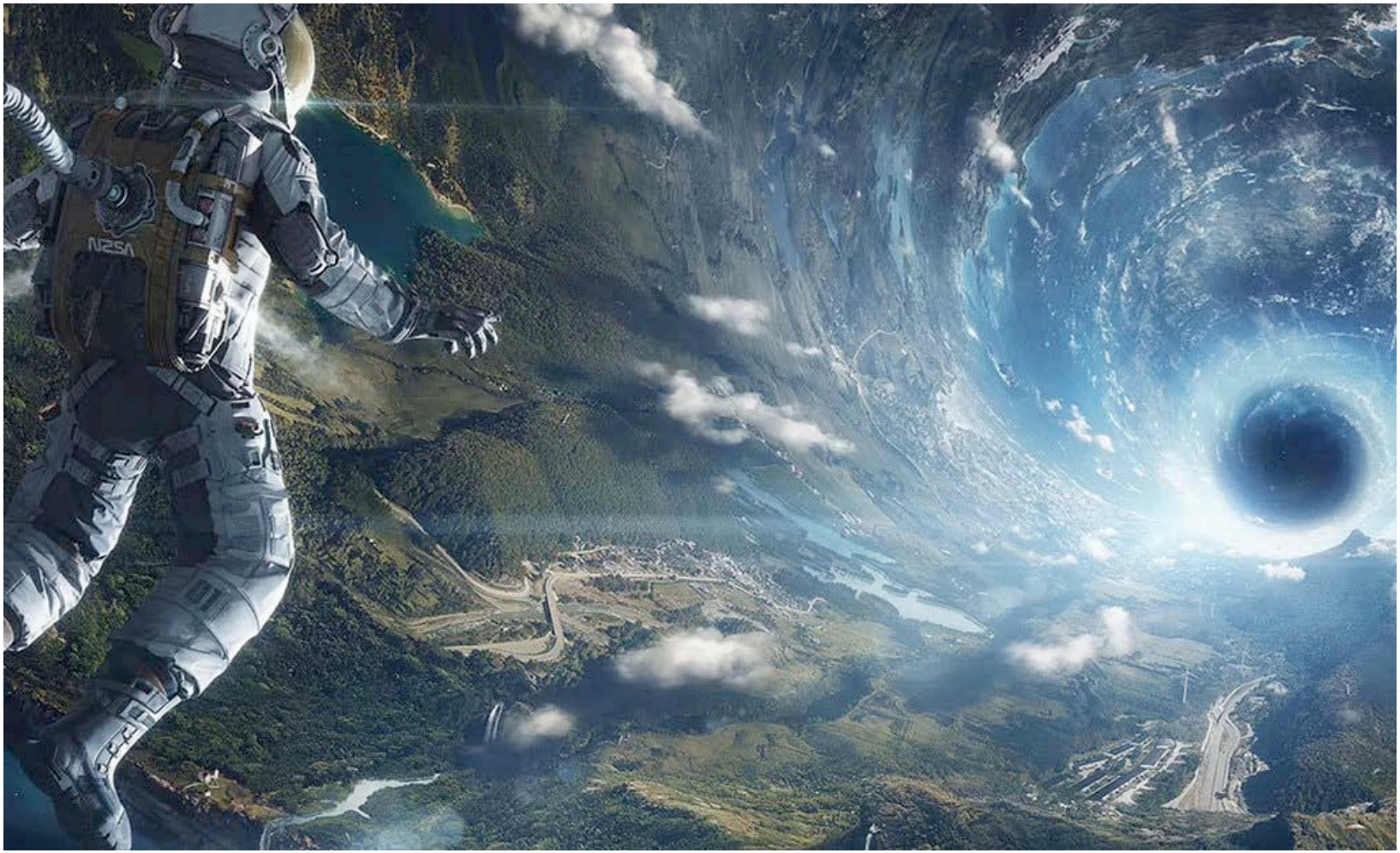
Ending Locations



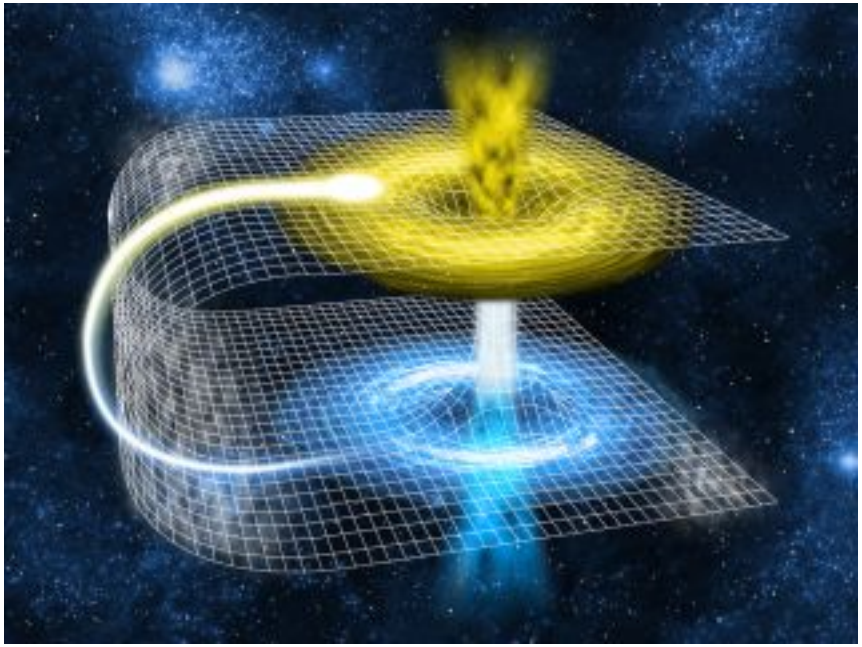
Ending Locations



Fields with Wormholes



Poll: Wormholes



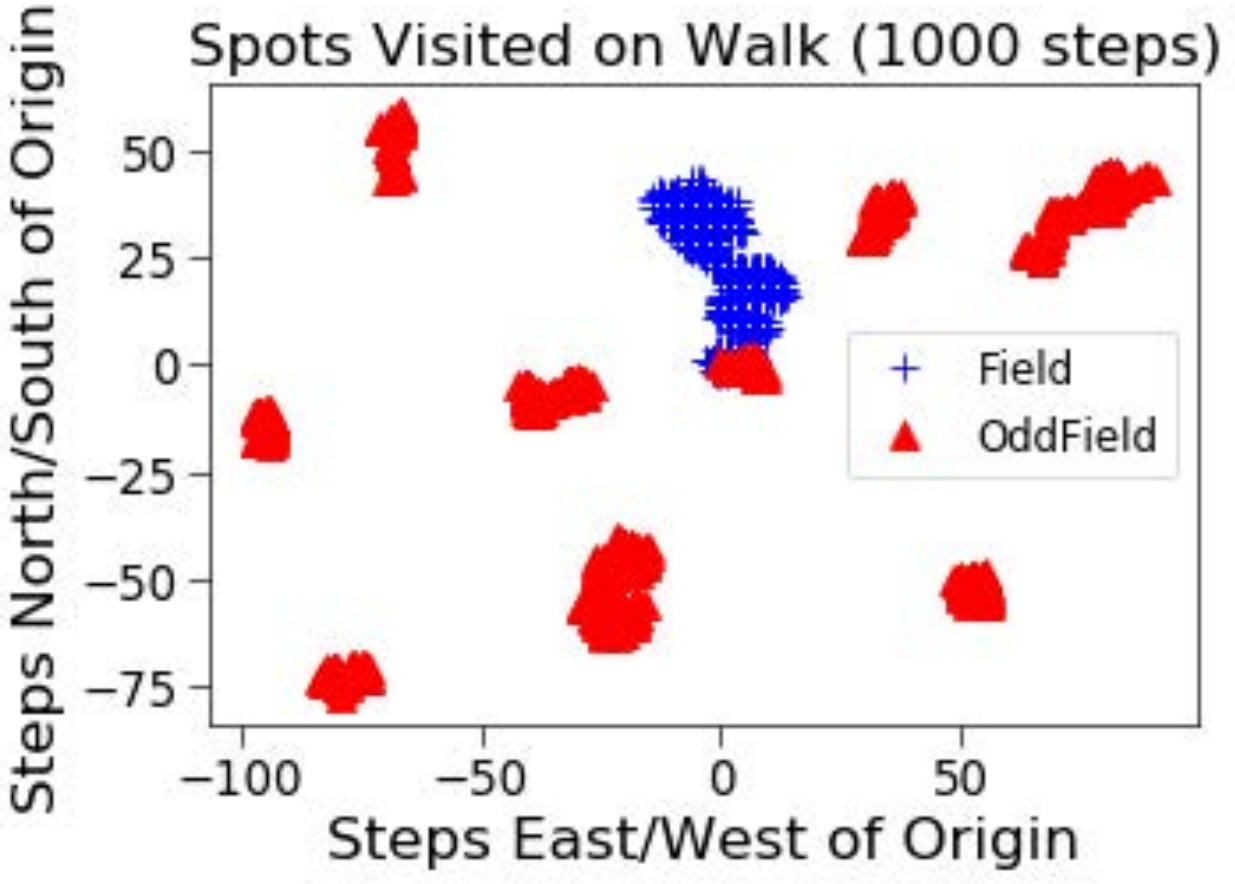
A Subclass of Field, part 1

```
class OddField(Field):
    def __init__(self, numHoles = 1000,
                 xRange = 100, yRange = 100):
        Field.__init__(self)
        self.wormholes = {}
        for w in range(numHoles):
            x = random.randint(-xRange, xRange)
            y = random.randint(-yRange, yRange)
            newX = random.randint(-xRange, xRange)
            newY = random.randint(-yRange, yRange)
            newLoc = Location(newX, newY)
            self.wormholes[(x, y)] = newLoc
```

A Subclass of Field, part 2

```
def moveDrunk(self, drunk):
    Field.moveDrunk(self, drunk)
    x = self.drunks[drunk].getX()
    y = self.drunks[drunk].getY()
    if (x, y) in self.wormholes:
        self.drunks[drunk] = self.wormholes[(x, y)]
```

Spots Reached During One Walk



Summary

- Point is not the simulations themselves, but how we built, evaluated, and used them
- Started by defining classes
- Built functions corresponding to
  - One trial, multiple trials, result reporting
- Made series of incremental changes to simulation so that we could investigate different questions
  - Get simple version working first
  - Did a sanity check!
  - Enhanced simulation a step at a time
- Showed how to use plots to get insights