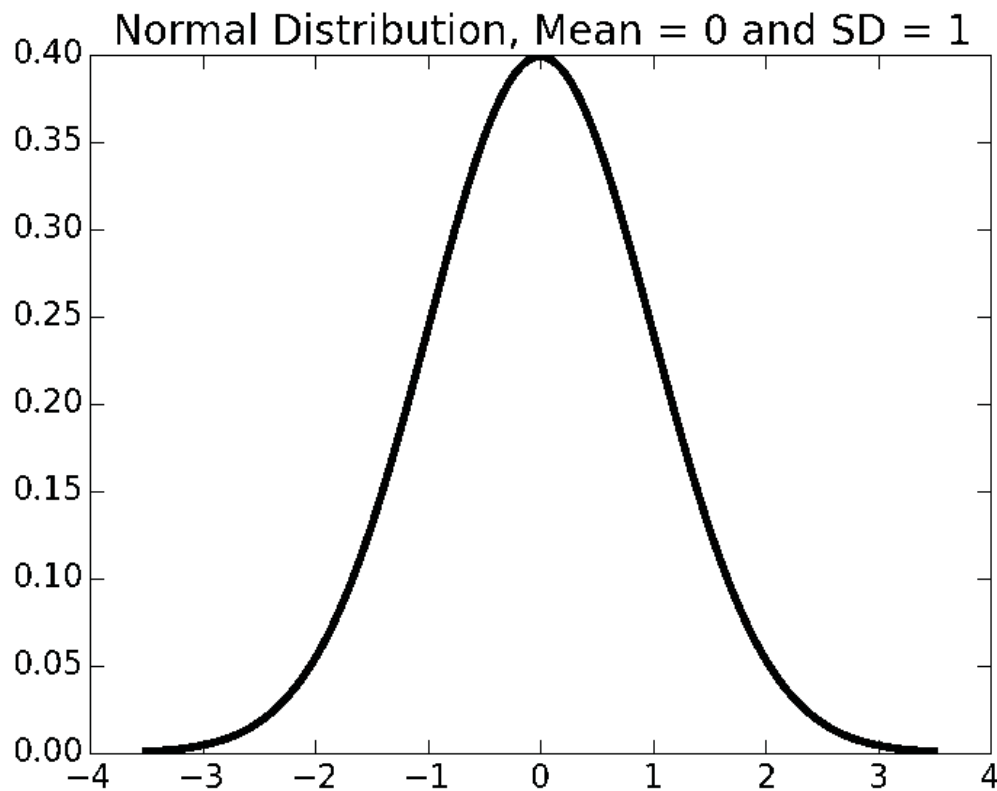


Lecture 7: Confidence Intervals

Assumptions Underlying Empirical Rule

- The mean estimation error is zero
- The distribution of the errors in the estimates is normal (Gaussian)

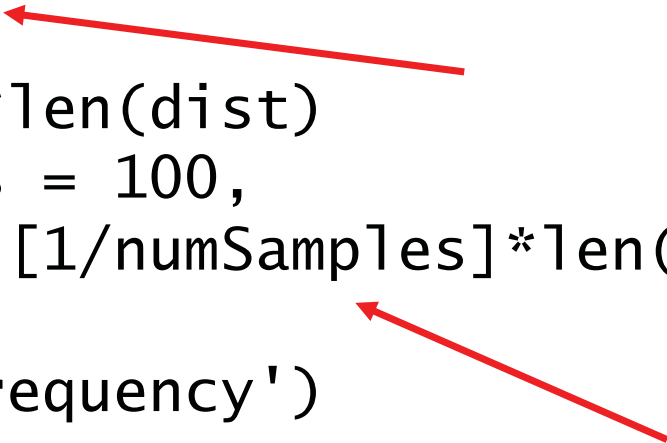


Generating Normally Distributed Data

```
dist, numSamples = [], 1000000
```

```
for i in range(numSamples):  
    dist.append(random.gauss(0, 100))
```

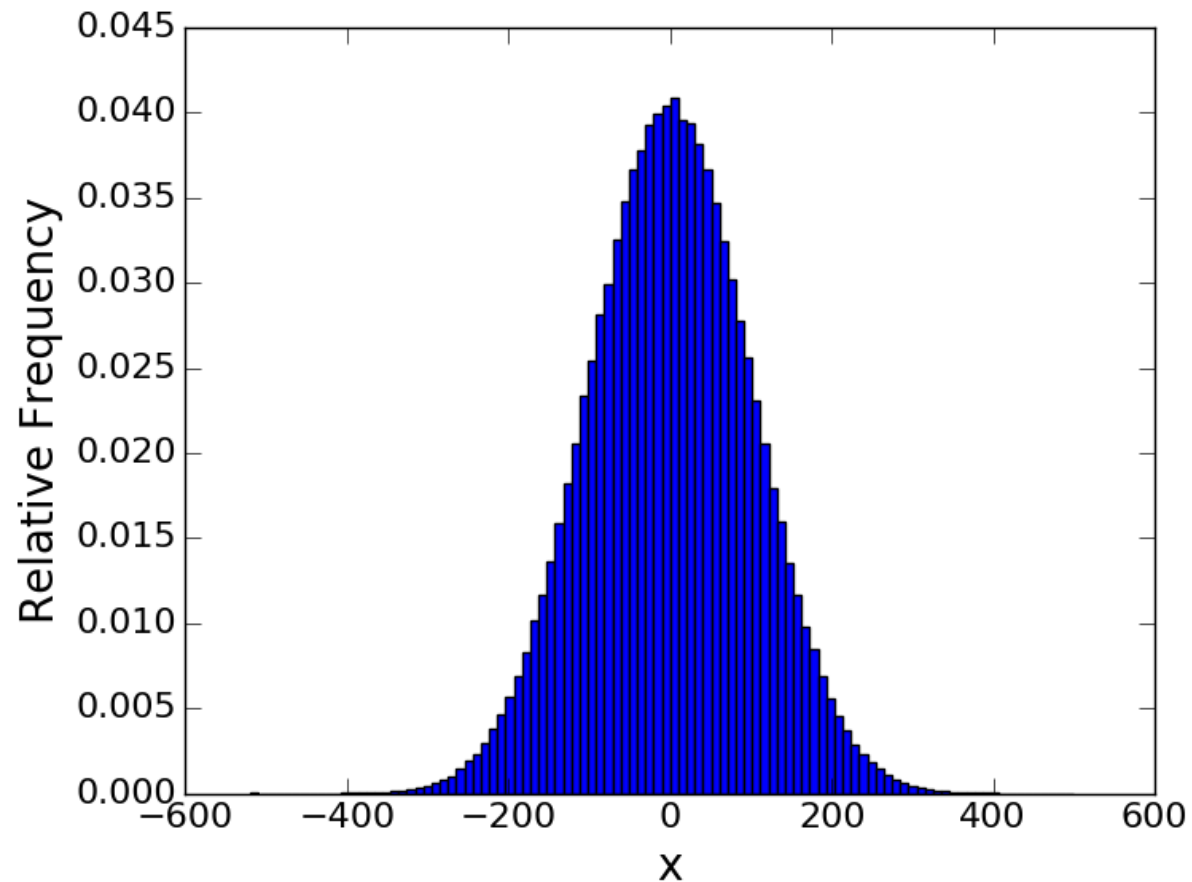
```
weights = [1/numSamples]*len(dist)  
v = pylab.hist(dist, bins = 100,  
               weights = [1/numSamples]*len(dist))  
pylab.xlabel('x')  
pylab.ylabel('Relative Frequency')
```



```
print('Fraction within ~200 of mean =',  
      sum(v[0][30:70]))
```

Output

Discrete
Approximation
to PDF



Fraction within ~ 200 of mean = 0.957147

PDF's (recapping)

- Distributions defined by *probability density functions* (PDFs)
- Probability of a random variable lying between two values
- Defines a curve where the values on the x-axis lie between minimum and maximum value of the variable
- Area under curve between two points, is probability of example falling within that range

PDF for Normal Distribution

```
def gaussian(x, mu, sigma):  
    factor1 = (1.0/(sigma*((2*pylab.pi)**0.5)))  
    factor2 = pylab.e**(-((x-mu)**2)/(2*sigma**2))  
    return factor1*factor2
```

```
xVals, yVals = [], []  
mu, sigma = 0, 1  
x = -4
```

```
while x <= 4:
```

```
    xVals.append(x)
```

```
    yVals.append(gaussian(x, mu, sigma))
```

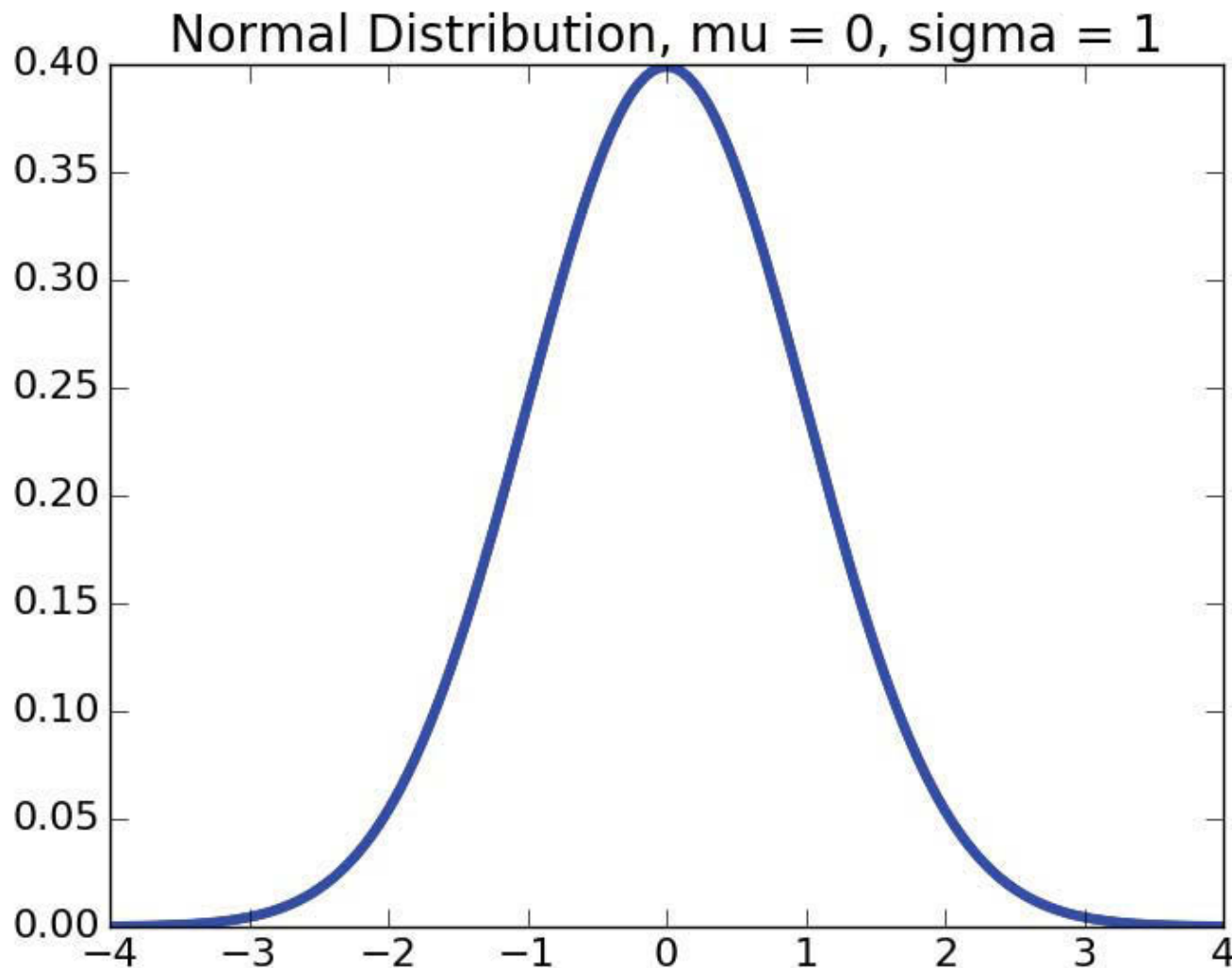
```
    x += 0.05
```

```
pylab.plot(xVals, yVals)
```

```
pylab.title('Normal Distribution, mu = ' + str(mu) \  
            + ', sigma = ' + str(sigma))
```

$$P(x) = \frac{1}{\sigma\sqrt{2\pi}} * e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

Output



Are values on y-axis probabilities?

They are densities.
I.e., derivative of cumulative distribution function.

Hence we use integration to interpret a PDF

A Digression

- **SciPy** library contains my useful mathematical functions used by scientists and engineers
- `scipy.integrate.quad` has up to four arguments
 - a function or method to be integrated
 - a number representing the lower limit of the integration,
 - a number representing the upper limit of the integration, and
 - an optional tuple supplying values for all arguments, except the first, of the function to be integrated
- `scipy.integrate.quad` returns a tuple
 - Approximation to result
 - Estimate of absolute error

Checking the Empirical Rule

```
import scipy.integrate ←
def gaussian(x, mu, sigma)
...
def checkEmpirical(numTrials):
    for t in range(numTrials):
        mu = random.randint(-10, 10)
        sigma = random.randint(1, 10)
        print('For mu =', mu, 'and sigma =', sigma)
        for numStd in (1, 1.96, 3):
            area = scipy.integrate.quad(gaussian,
                                         mu-numStd*sigma,
                                         mu+numStd*sigma,
                                         (mu, sigma))[0]
            print(' Fraction within', numStd,
                  'std =', round(area, 4))
```

Results

For $\mu = 9$ and $\sigma = 6$

Fraction within 1 std = 0.6827

Fraction within 1.96 std = 0.95

Fraction within 3 std = 0.9973

For $\mu = -6$ and $\sigma = 5$

Fraction within 1 std = 0.6827

Fraction within 1.96 std = 0.95

Fraction within 3 std = 0.9973

For $\mu = 2$ and $\sigma = 6$

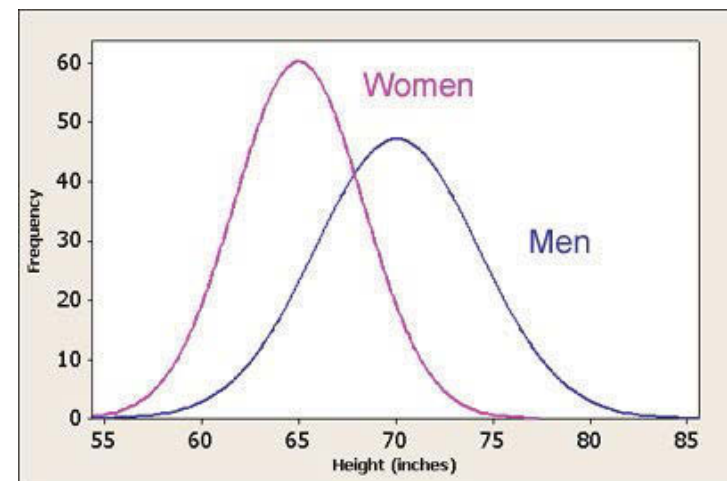
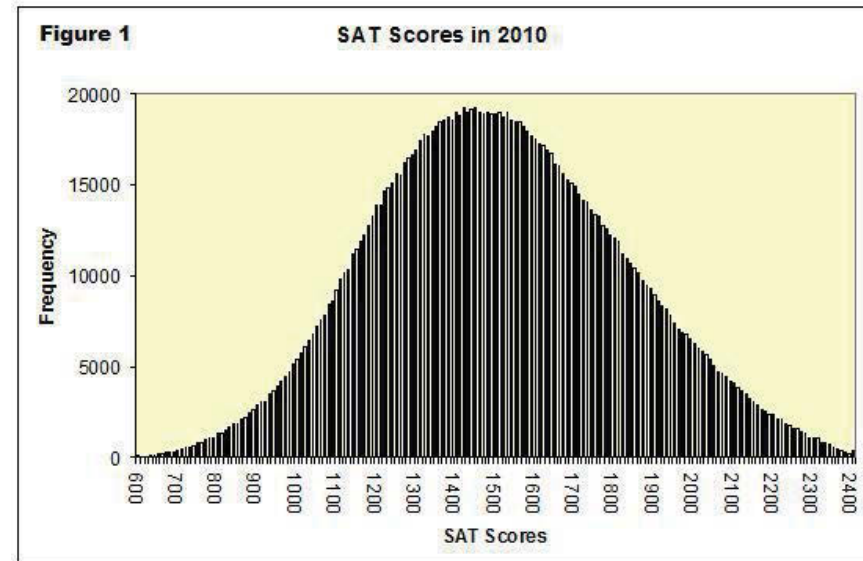
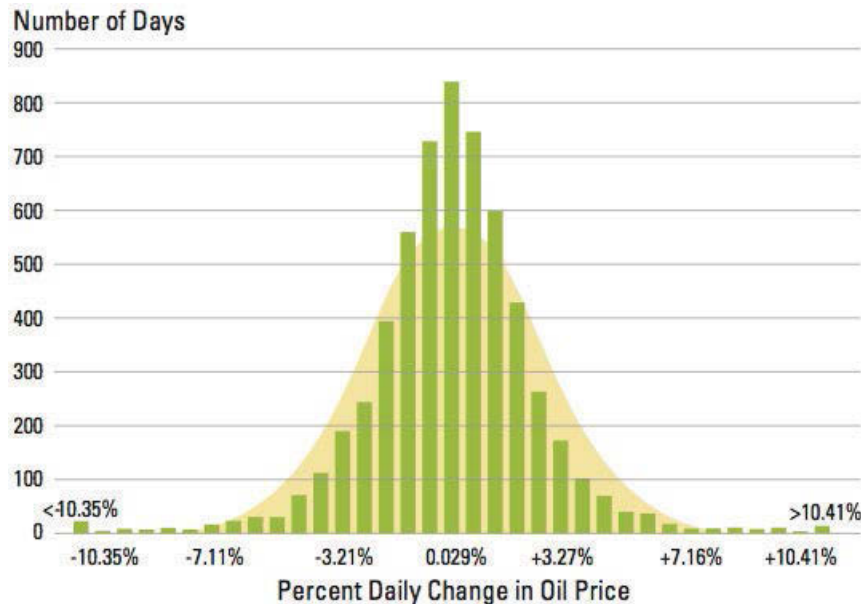
Fraction within 1 std = 0.6827

Fraction within 1.96 std = 0.95

Fraction within 3 std = 0.9973

Everybody Likes Normal Distributions

- Occur a lot!
- Nice mathematical properties

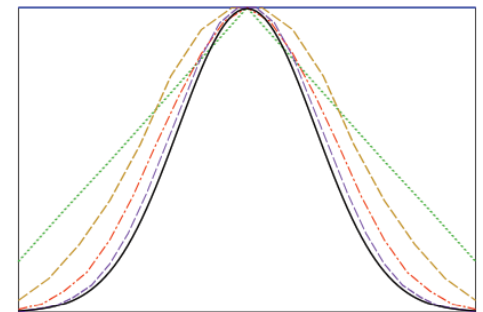
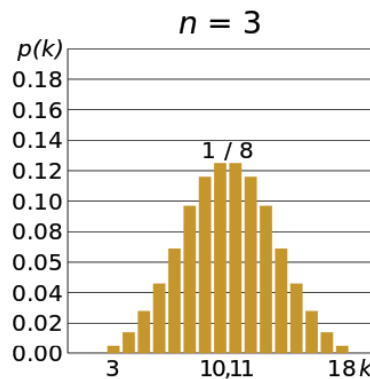
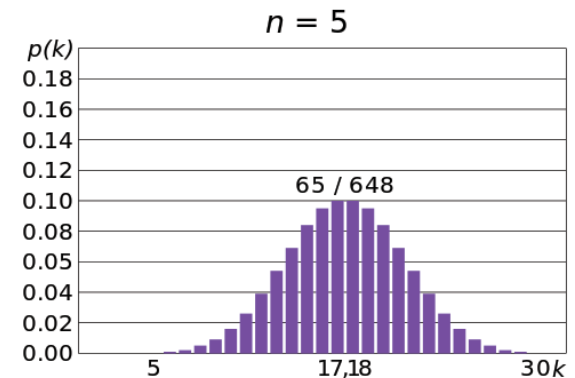
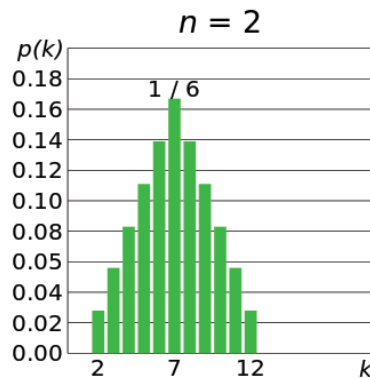
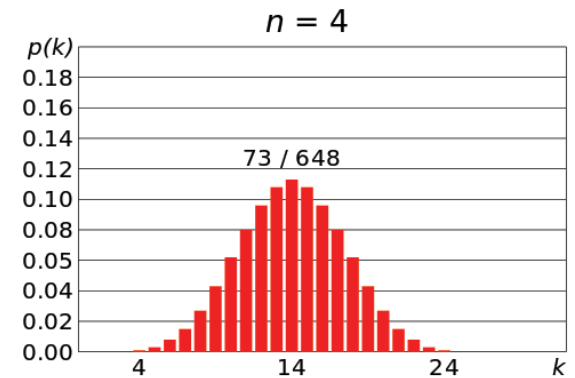
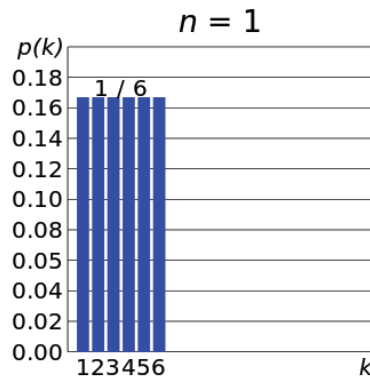


But Not All Distribution Are Normal

- Empirical works for normal distributions
- But are the outcomes of spins of a roulette wheel normally distributed?
- No, they are uniformly distributed
 - Each outcome is equally probable
- So, why does the empirical rule work here?

Why Did the Empirical Rule Work?

- Because we are reasoning not about a single spin, but about the mean of a set of spins
- And the **central limit theorem** applies



The Central Limit Theorem (CLT)

- Given a sufficiently large sample:
 - 1) The means of the samples in a set of samples (the sample means) will be approximately normally distributed,
 - 2) This normal distribution will have a mean close to the mean of the population, and
 - 3) The variance of the sample means will be close to the variance of the population divided by the sample size.

Checking CLT for a Continuous Die

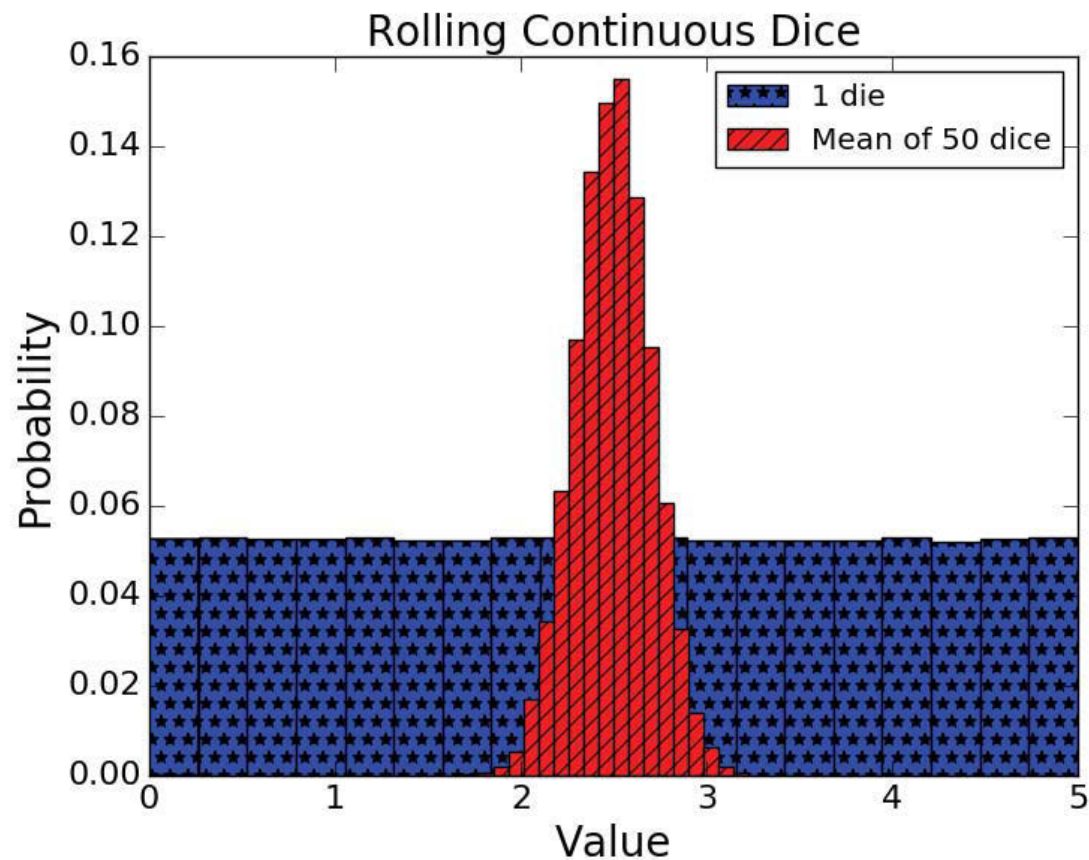
```
def plotMeans(numDice, numRolls, numBins, legend, color, style):
    means = []
    for i in range(numRolls//numDice):
        vals = 0
        for j in range(numDice):
            vals += 5*random.random()
        means.append(vals/float(numDice))
    pylab.hist(means, numBins, color = color, label = legend,
    → weights = pylab.array(len(means)*[1])/len(means),
        hatch = style)
    return getMeanAndStd(means)

mean, std = plotMeans(1, 1000000, 19, '1 die', 'b', '*')
print('Mean of rolling 1 die =', str(mean) + ', ', 'Std =', std)
mean, std = plotMeans(50, 1000000, 19, 'Mean of 50 dice', 'r', '//')
print('Mean of rolling 50 dice =', str(mean) + ', ', 'Std =', std)
pylab.title('Rolling Continuous Dice')
pylab.xlabel('Value')
pylab.ylabel('Probability')
pylab.legend()
```

Output

Mean of rolling 1 die = 2.49759575528, Std = 1.4439045633

Mean of rolling 50 dice = 2.49985051798, Std = 0.204887274645



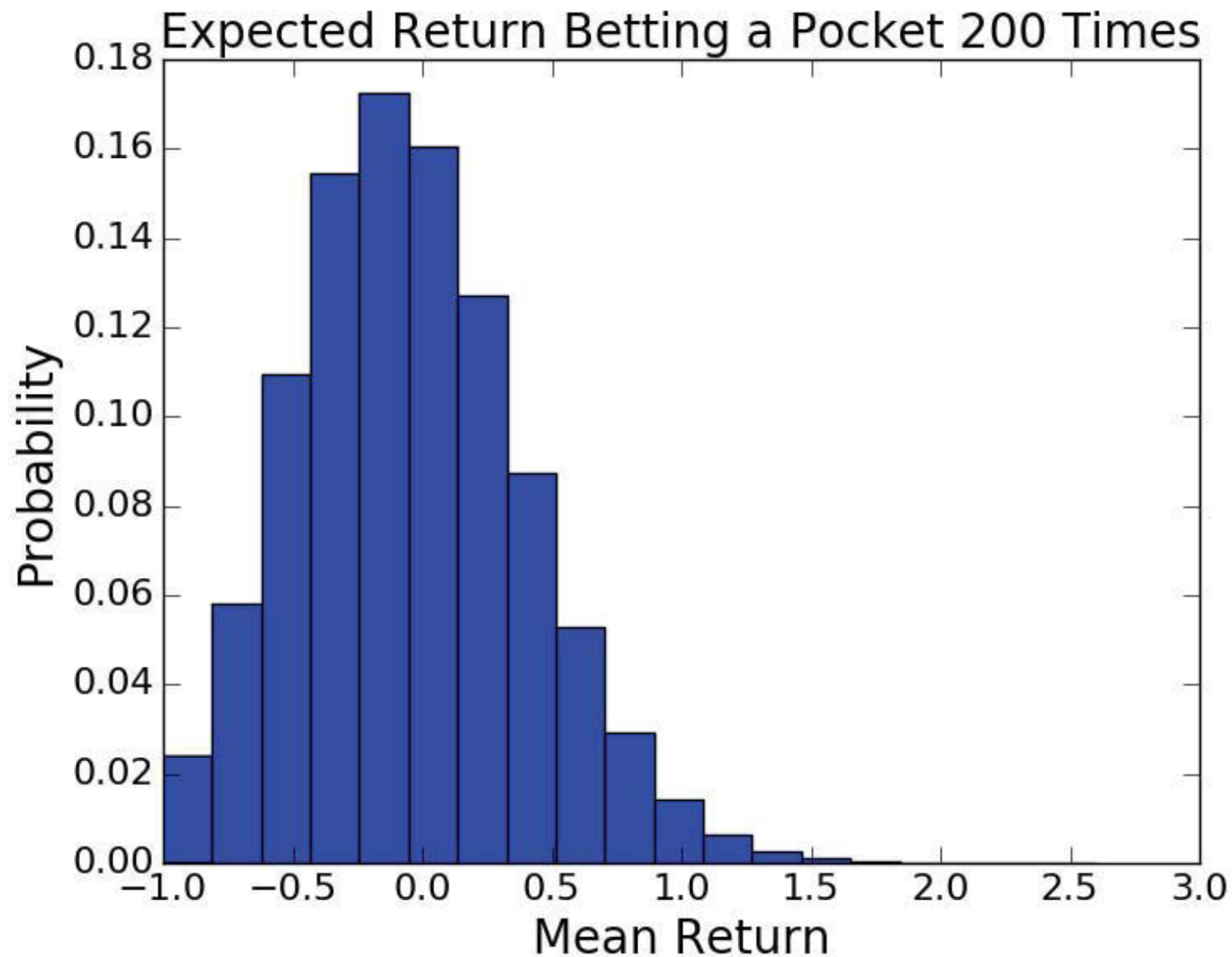
Try It for Roulette

```
numTrials = 1000000
numSpins = 200
game = FairRoulette()

means = []
for i in range(numTrials):
    means.append(findPocketReturn(game, 1, numSpins,
                                   False)[0])

pylab.hist(means, bins = 19,
            weights = [1/len(means)]*len(means))
pylab.xlabel('Mean Return')
pylab.ylabel('Probability')
pylab.title('Expected Return Betting a Pocket 200 Times')
```

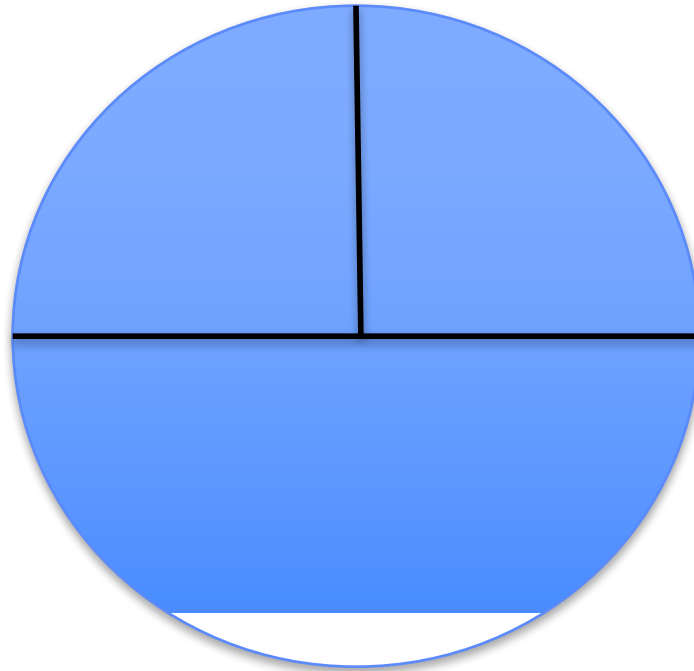
Betting a Pocket in Fair Roulette



Moral

- It doesn't matter what the shape of the distribution of values happens to be
- If we are trying to estimate the mean of a population using sufficiently large samples
- The CLT allows us to use the empirical rule when computing confidence intervals

Pi



$$\frac{\textit{circumference}}{\textit{diameter}} = \Pi \quad \textit{area} = \Pi * \textit{radius}^2$$

Rhind Papyrus



$$4 \cdot (8/9)^2 = 3.16$$

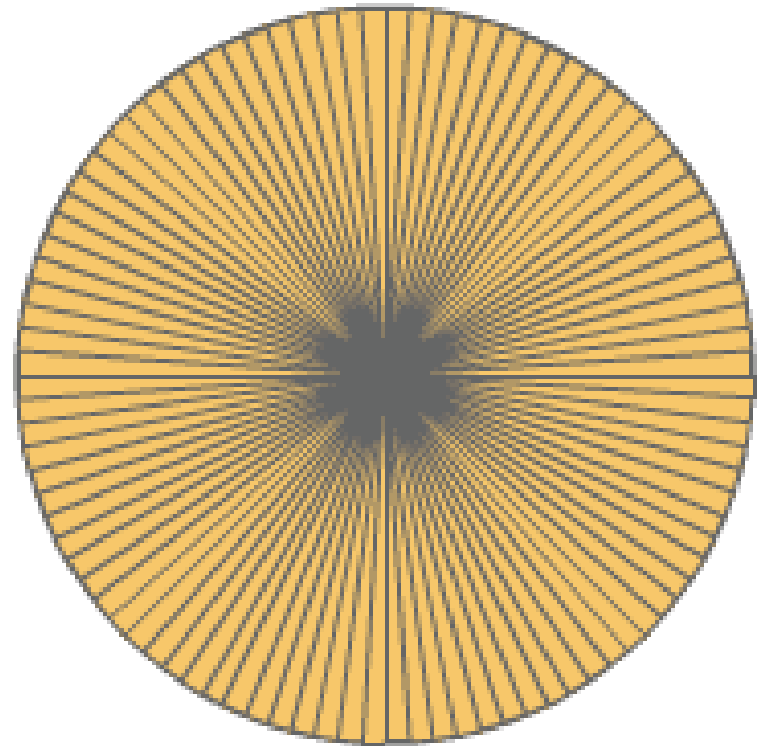
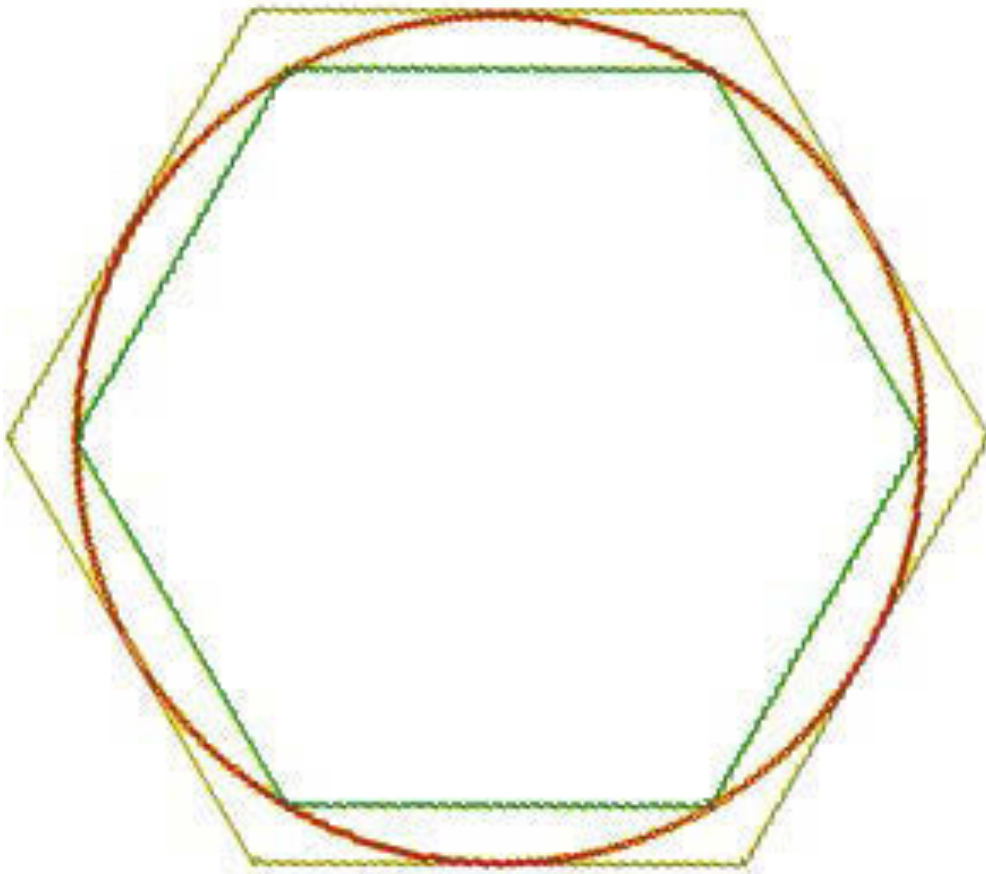
Image of the Rhind Papyrus is in the public domain. Source: [Wikimedia Commons](#).

~1100 Years Later

“And he made a molten sea, ten cubits from the one brim to the other: it was round all about, and his height was five cubits: and a line of thirty cubits did compass it round about.”

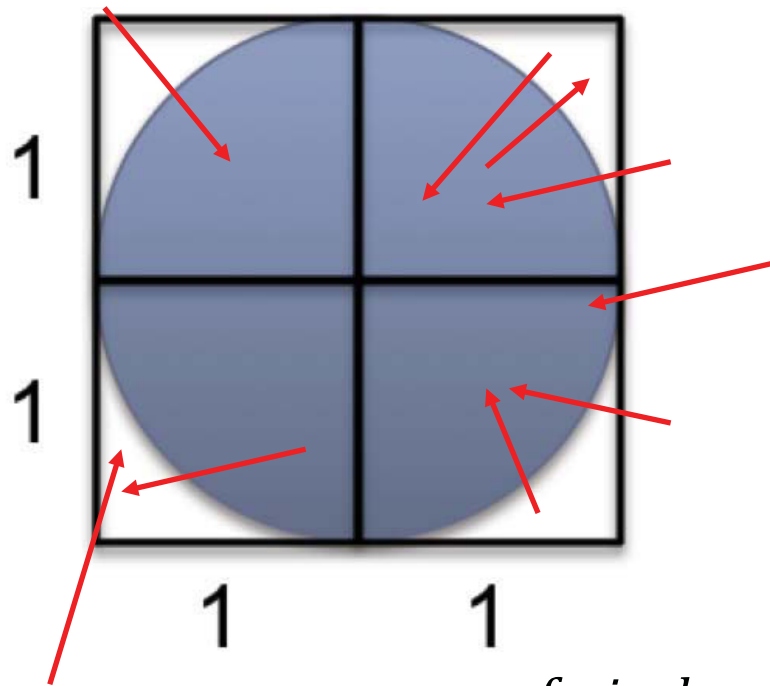
—1 Kings 7.23

~300 Years Later (Archimedes)



$$223/71 < \pi < 22/7$$

~2000 Years Later (Buffon-Laplace)



$$A_s = 2 * 2 = 4$$

$$A_c = \pi r^2 = \pi$$

$$\frac{\text{needles in circle}}{\text{needles in square}} = \frac{\text{area of circle}}{\text{area of square}}$$

$$\text{area of circle} = \frac{\text{area of square} * \text{needles in circle}}{\text{needles in square}}$$

$$\text{area of circle} = \frac{4 * \text{needles in circle}}{\text{needles in square}}$$

~200 Years Later



Crazy archer on closed course. Do not try ANYWHERE.

<https://www.youtube.com/watch?v=oYM6MljZ8IY>

Very End of Video



Simulating Buffon-Laplace Method

```
def throwNeedles(numNeedles):  
    inCircle = 0  
    for Needles in range(1, numNeedles + 1, 1):  
        x = random.random()  
        y = random.random()  
        if (x*x + y*y)**0.5 <= 1.0:  
            inCircle += 1  
    return 4*(inCircle/float(numNeedles))
```

Simulating Buffon-Laplace Method, cont.

```
def getEst(numNeedles, numTrials):
    estimates = []
    for t in range(numTrials):
        piGuess = throwNeedles(numNeedles)
        estimates.append(piGuess)
    sDev = stdDev(estimates)
    curEst = sum(estimates)/len(estimates)
    print('Est. = ' + str(curEst) + \
          ', Std. dev. = ' + str(round(sDev, 6)) \
          + ', Needles = ' + str(numNeedles))
    return (curEst, sDev)
```

Simulating Buffon-Laplace Method, cont.

```
def estPi(precision, numTrials):  
    numNeedles = 1000  
    sDev = precision  
    while sDev >= precision/2:  
        curEst, sDev = getEst(numNeedles,  
                               numTrials)  
        numNeedles *= 2  
    return curEst  
  
estPi(0.005, 100)
```

Output

Est. = 3.1484400000000012, Std. dev. = 0.047886, Needles = 1000
Est. = 3.1391799999999987, Std. dev. = 0.035495, Needles = 2000
Est. = 3.1410799999999997, Std. dev. = 0.02713, Needles = 4000
Est. = 3.141435, Std. dev. = 0.016805, Needles = 8000
Est. = 3.141355, Std. dev. = 0.0137, Needles = 16000
Est. = 3.1413137500000006, Std. dev. = 0.008476, Needles = 32000
Est. = 3.141171874999999, Std. dev. = 0.007028, Needles = 64000
Est. = 3.1415896874999993, Std. dev. = 0.004035, Needles = 128000
Est. = 3.1417414062499995, Std. dev. = 0.003536, Needles = 256000
Est. = 3.14155671875, Std. dev. = 0.002101, Needles = 512000

Being Right is Not Good Enough

- Not sufficient to produce a good answer
- Need to have reason to believe that it is close to right
- In this case, small standard deviation implies that we are close to the true value of π

Right?

Is it Correct to State

- 95% of the time we run this simulation, we will estimate that the value of π is between 3.13743875875 and 3.14567467875?
- With a probability of 0.95 the actual value of π is between 3.13743875875 and 3.14567467875?
- Both are factually correct
- But only one of these statement can be inferred from our simulation
- *statisically valid* \neq *true*

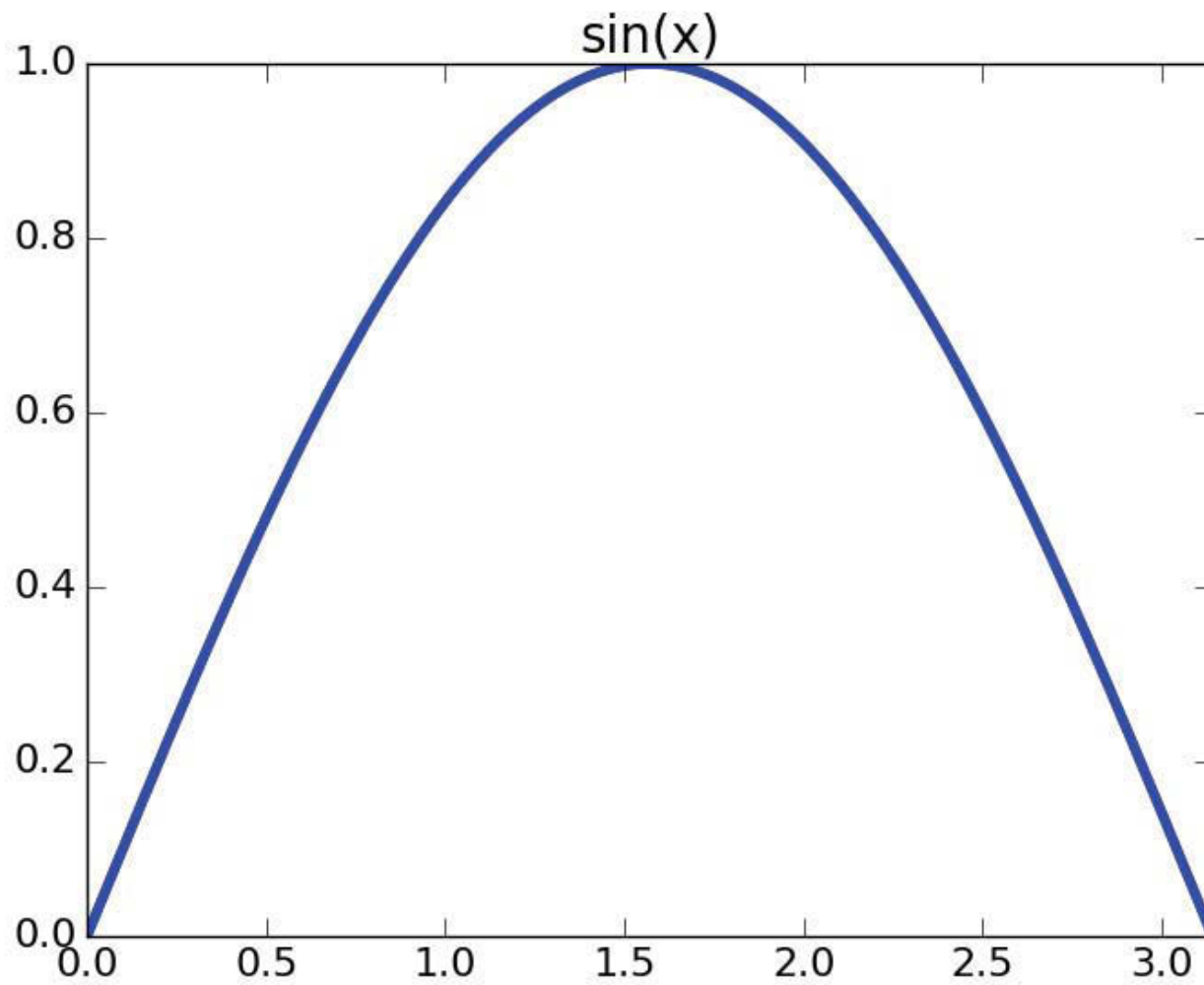
Introduce a Bug

```
def throwNeedles(numNeedles):  
    inCircle = 0  
    for Needles in range(1, numNeedles + 1, 1):  
        x = random.random()  
        y = random.random()  
        if (x*x + y*y)**0.5 <= 1.0:  
            inCircle += 1  
    return 2*(inCircle/float(numNeedles))
```

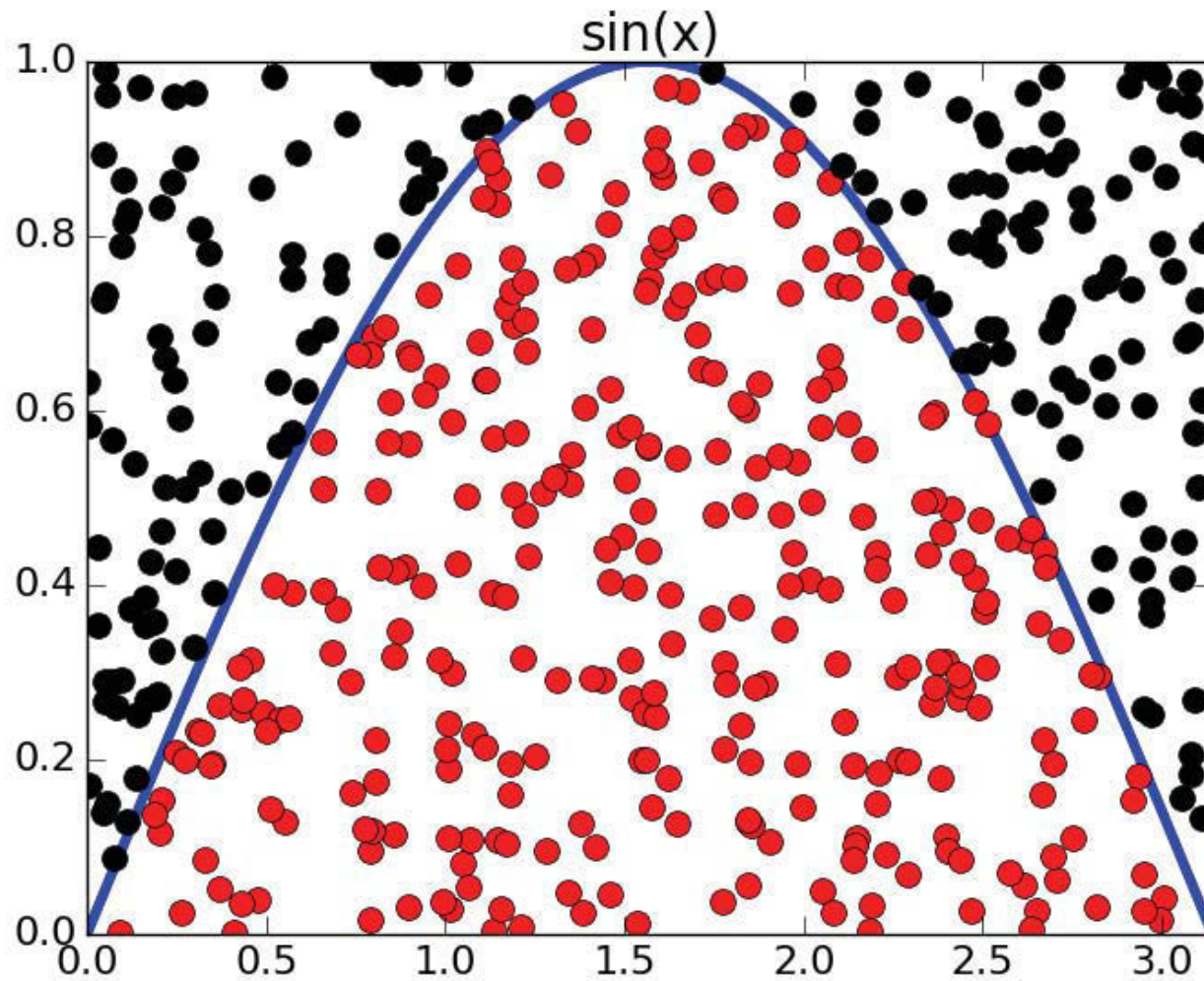
Generally Useful Technique

- To estimate the area of some region, R
 - Pick an enclosing region, E , such that the area of E is easy to calculate and R lies completely within E
 - Pick a set of random points that lie within E
 - Let F be the fraction of the points that fall within R
 - Multiply the area of E by F
- Way to estimate integrals

Sin(x)



Random Points



MIT OpenCourseWare

<https://ocw.mit.edu>

6.0002 Introduction to Computational Thinking and Data Science

Fall 2016

For information about citing these materials or our Terms of Use, visit: <https://ocw.mit.edu/terms>.