Randomness

What type of problems can we solve with the help of random numbers?

We can compute (potentially) complicated averages:

- How much my stock/option portfolio is going to be worth?
- What are my odds to win a certain competition?

Random Number Generators

- Computers are deterministic - operations are reproducible
- How do we get random numbers out of a deterministic machine?

In [1]:
```
import numpy as np
```

In [2]:
```
np.random.rand(10)
```

Out[2]:
```
array([0.19252561, 0.95875038, 0.34205705, 0.44060429, 0.28911005,
       0.82233359, 0.29041589, 0.29592469, 0.69701575, 0.81690982])
```

In [3]:
```
for x in range(0, 20):
    numbers = np.random.rand(6)
    #print(numbers)
```

They all seem random correct? Let's try to fix something called **seed** using np.random.seed(10)

What do you observe?

Let's see what this seed is...

Pseudo-random Numbers

- Numbers and sequences appear random, but they are actually reproducible
- Great for algorithm developing and debugging
- How truly "random" are these numbers?

Linear congruential generator

Given the parameters a, c, m and s, where s is the seed value, this algorithm will generate a sequence of pseudo-random numbers:

$x_o = s$

$x_{n+1} = (ax_n + c) mod(m)$

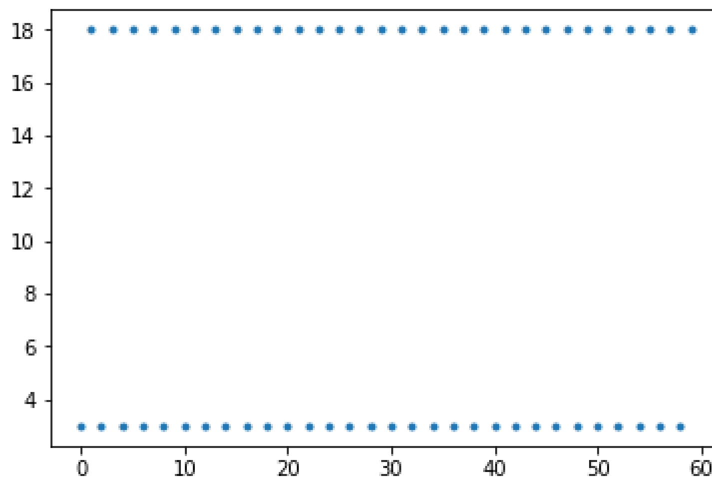
In [4]:
```
s = 3 # seed
a = 37 #10 # multiplier
c = 2 # increment
m = 19 # modulus
```

In [5]:
```
n = 60
x = np.zeros(n)
x[0] = s
for i in range(1,n):
    x[i] = (a * x[i-1] + c) % m
```

In [6]:
```
import matplotlib.pyplot as plt
%matplotlib inline
```

In [7]:
```
plt.plot(x,'.')
```

Out[7]:
```
[<matplotlib.lines.Line2D at 0x11227fd68>]
```



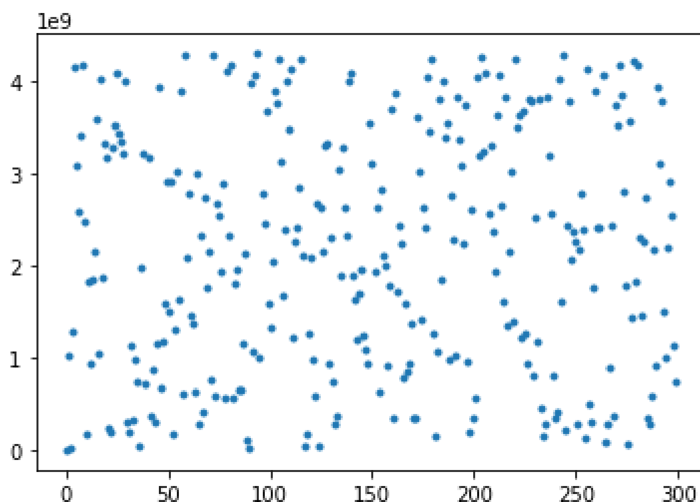
Notice there is a period, when numbers eventually start repeating. One of the advantages of the LCG is that by using appropriate choice for the parameters, we can obtain known and long periods.

Check here https://en.wikipedia.org/wiki/Linear_congruential_generator (https://en.wikipedia.org/wiki/Linear_congruential_generator) for a list of commonly used parameters of LCGs.

Let's try using the parameters from Numerical recipes (https://en.wikipedia.org/wiki/Numerical_Recipes)

In [8]:
```
s = 8
a = 1664525
c = 1013904223
m = 2**32

n = 300
x = np.zeros(n)
x[0] = s
for i in range(1,n):
    x[i] = (a * x[i-1] + c) % m

plt.plot(x,'.')
```

Out[8]:
```
[<matplotlib.lines.Line2D at 0x11237add8>]
```



"Good" random number generators are efficient, have long peiods and are portable.

Random Variables

Think of a random variable X as a function that maps the outcome of an unpredictable (random) processses to numerical quantities.

For example:

- X = the face of a bread when it falls on the ground. The random value can abe the "buttered" side or the "not buttered" side
- X = value that appears on top of dice after each roll

We don't have an exact number to represent these random processes, but we can get something that represents the **average** case. To do that, we need to know the likelihood of each individual value of X.

Coin toss

Random variable X: result of a coin toss

In each toss, the random variable can take the values $x_1 = 0$ (tail) and $x_2 = 1$ (head), and each x_i has probability $p_i = 0.5$.

The **expected value** of a discrete random variable is defined as:

$$E(x) = \sum_{i=1}^{m} p_i x_i$$

Hence for a coin toss we have:

$$E(x) = 1(0.5) + 0(0.5) = 0.5$$

Roll Dice

Random variable X: value that appears on top of the dice after each roll

In each toss, the random variable can take the values $x_i = 1, 2, 3, \ldots, 6$ and each x_i has probability $p_i = 1/6$.

The **expected value** of the discrete random variable is defined as:

In [9]:
```
E = 0
for i in range(6):
    E += (i+1)*(1/6)
E
```

Out[9]: 3.5

Monte Carlo Methods

Monte Carlo methods are algorithms that rely on repeated random sampling to approximate a desired quantity.

Simulating a coin toss experiment

We want to find the probability of heads when tossing a coin. We know the expected value is 0.5. Using Monte Carlo with N samples (here tosses), our estimate of the expected value is:

$$E = \frac{1}{N}\sum_{i=1}^{N} f(x_i) = \frac{1}{N}\sum_{i=1}^{N} x_i$$

where $x_i = 1$ if the toss gives head.

Let's toss a "fair" coin N times and record the results for each toss.

But first, how can we simulate one toss?

In [10]:
```
toss = np.random.choice([0,1])
print(toss)
```

```
0
```

In [11]:
```
N = 30 # number of samples (tosses)

toss_list = []
for i in range(N):
    toss = np.random.choice([0,1])
    toss_list.append(toss)

np.array(toss_list).sum()/N
```

Out[11]:
```
0.5
```

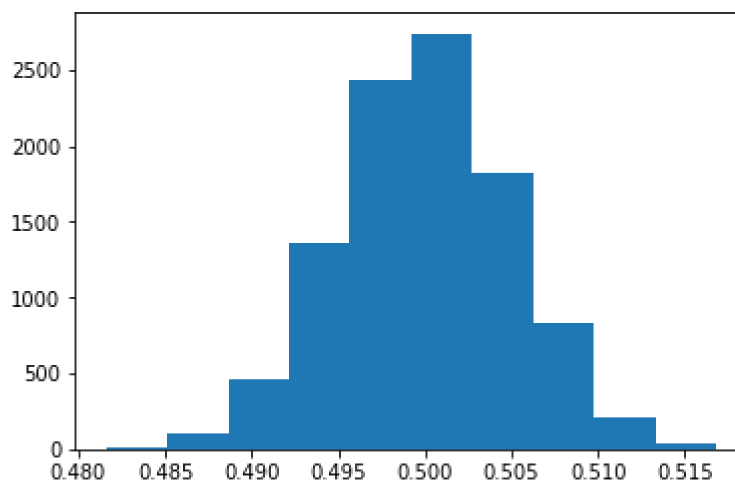
Note that if we run the code snippet above again, it is likely we will get a different result. What if we run this many times?

In [37]:

```
#clear
N = 10000 # number of tosses
M = 10000 # number of numerical experiments
nheads = []
for j in range(M):
    toss_list = []
    for i in range(N):
        toss_list.append(np.random.choice([0,1]))
    nheads.append( np.array(toss_list).sum()/N )
nheads = np.array(nheads)

plt.hist(nheads)
```

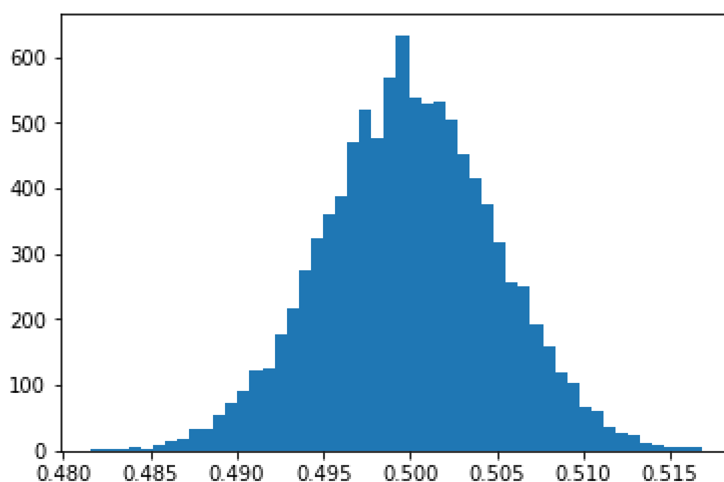
Out[37]:

```
(array([  14.,  109.,  464., 1355., 2425., 2738., 1821.,  826.,  213.,
          35.]),
 array([0.4816 , 0.48512, 0.48864, 0.49216, 0.49568, 0.4992 , 0.50272,
        0.50624, 0.50976, 0.51328, 0.5168 ]),
 <a list of 10 Patch objects>)
```



In [38]:
```
plt.hist(nheads, bins=50)
```

Out[38]:
```
(array([  2.,   1.,   3.,   5.,   3.,   8.,  14.,  19.,  34.,  34.,  5
5.,
         72.,  92., 121., 124., 176., 218., 275., 324., 362., 388., 47
2.,
        519., 478., 568., 634., 539., 530., 531., 504., 451., 417., 37
7.,
        319., 257., 251., 191., 160., 119., 105.,  66.,  60.,  35.,  2
7.,
         25.,  11.,  10.,   5.,   4.,   5.]),
 array([0.4816  , 0.482304, 0.483008, 0.483712, 0.484416, 0.48512 ,
        0.485824, 0.486528, 0.487232, 0.487936, 0.48864 , 0.489344,
        0.490048, 0.490752, 0.491456, 0.49216 , 0.492864, 0.493568,
        0.494272, 0.494976, 0.49568 , 0.496384, 0.497088, 0.497792,
        0.498496, 0.4992  , 0.499904, 0.500608, 0.501312, 0.502016,
        0.50272 , 0.503424, 0.504128, 0.504832, 0.505536, 0.50624 ,
        0.506944, 0.507648, 0.508352, 0.509056, 0.50976 , 0.510464,
        0.511168, 0.511872, 0.512576, 0.51328 , 0.513984, 0.514688,
        0.515392, 0.516096, 0.5168  ]),
 <a list of 50 Patch objects>)
```



In [34]:
```
print(nheads.mean(),nheads.std())
```

```
0.4996731 0.005044558096602714
```

What happens when we increase the number of numerical experiments?

Approximating integrals

To approximate an integration

$$A = \int_{x_1}^{x_2} \int_{y_1}^{y_2} f(x,y)dxdy$$

we sample points uniformily inside a domain $D = [x_1, x_2] \times [y_1, y2]$, i.e. we let X be a uniformily distributed random variable on D. Using Monte Carlo with N sample points, our estimate for the expected value (that a sample point is inside the circle) is:

$$S_N = \frac{1}{N} \sum_{i=1}^{N} f(X_i)$$

which gives the approximate for the integral:

$$A_N = (x_2 - x_1)(y_2 - y_1) \frac{1}{N} \sum_{i=1}^{N} f(X_i)$$

Law of large numbers:

as $N \to \infty$, the sample average S_N converges the the expected value $E(X)$ and hence $A_N \to A$

We will use Monte Carlo Method to approximate the area of a circle of radius R = 1.

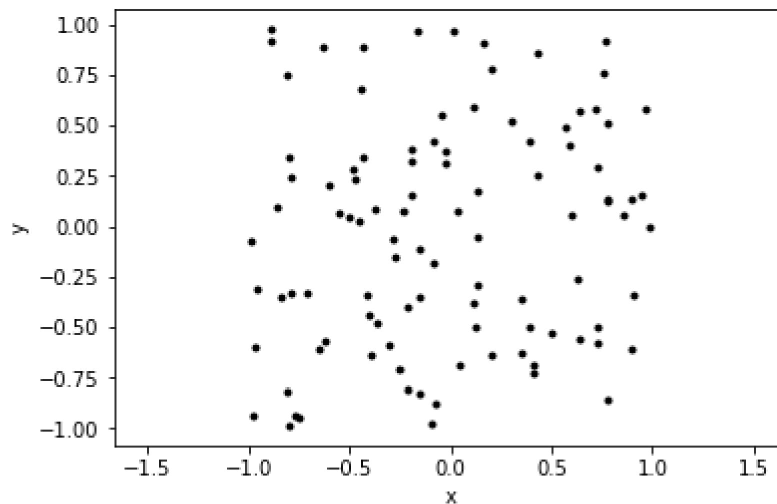
Let's start with a uniform distribution on the unit square [0,1]×[0,1] . Create a 2D array samples of shape (2, N):

In [15]:
```
N = 10**2
samples = np.random.rand(2, N)
```

Scale the sample points "samples", so that we have a uniform distribution inside a square $[-1, 1] \times [-1, 1]$. Calculate the distance from each sample point to the origin $(0, 0)$

In [16]:
```
xy = samples * 2 - 1.0 # scale sample points
r = np.sqrt(xy[0, :]**2 + xy[1, :]**2)  # calculate radius
```

In [17]:
```
plt.plot(xy[0,:], xy[1,:], 'k.')
plt.axis('equal')
plt.xlabel('x')
plt.ylabel('y');
```



We then count how many of these points are inside the circle centered at the origin.

In [18]:
```
incircle = (r <= 1)
count_incircle = incircle.sum()
print(count_incircle)
```

```
84
```

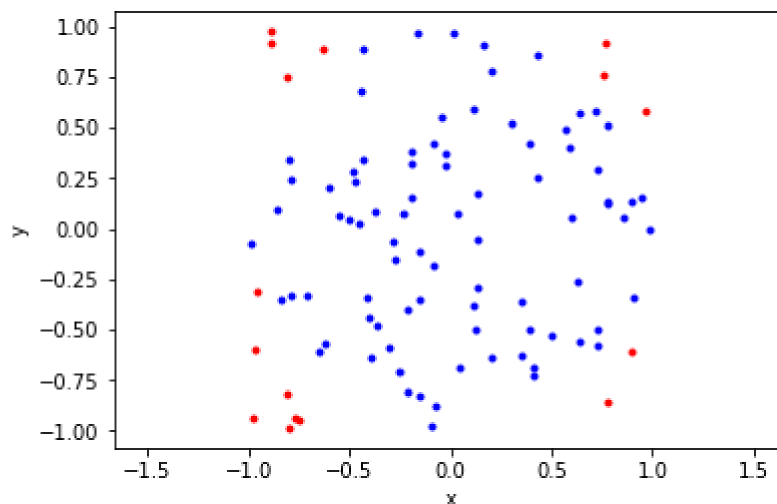
And the approximated value for the area is:

In [19]:
```
A_approx = (2*2) * (count_incircle)/N
A_approx
```

Out[19]: 3.36

We can assign different colors to the points inside the circle and plot (just for vizualization purposes).

In [20]:
```
plt.plot(xy[0,np.where(incircle)[0]], xy[1,np.where(incircle)[0]], 'b.')
plt.plot(xy[0,np.where(incircle==False)[0]], xy[1,np.where(incircle==False)[0]], 'r.')
plt.axis('equal')
plt.xlabel('x')
plt.ylabel('y');
```



Combine all the relevant code above, so we can easily run this numerical experiment for different sample size N.

In [21]:
```
#clear
N = 10**2
samples = np.random.rand(2, N)
xy = samples * 2 - 1.0 # scale sample points
r = np.sqrt(xy[0, :]**2 + xy[1, :]**2)  # calculate radius
incircle = (r <= 1)
count_incircle = incircle.sum()
A_approx = (2*2) * (count_incircle)/N
print(A_approx)
```

```
3.0
```

Perform the same above, but now store the approximated area for different N, and plot:

In [22]:
```
#clear
N = 10**6
samples = np.random.rand(2, N)
xy = samples * 2 - 1.0 # scale sample points
r = np.sqrt(xy[0, :]**2 + xy[1, :]**2)  # calculate radius
incircle = (r <= 1)
N_samples = np.arange(1,N+1)
A_approx = 4 * incircle.cumsum() / N_samples
```

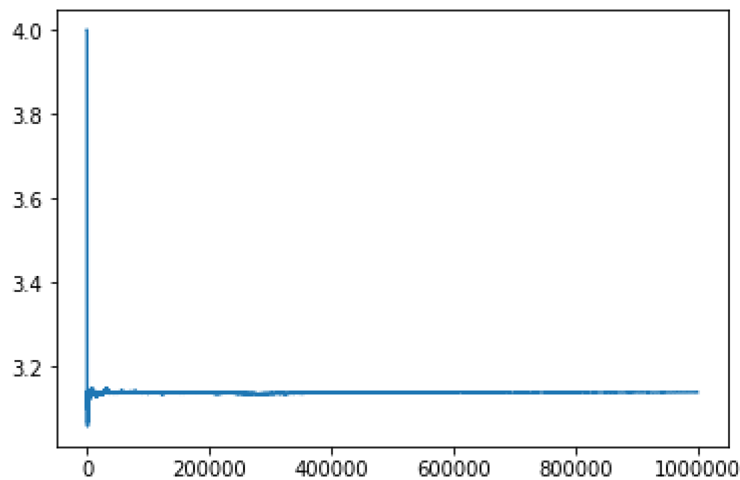
The approximated area is:

In [23]:
```
#clear
A_approx[-1]
```

Out[23]: 3.140908

In [24]:
```
plt.plot(A_approx)
```

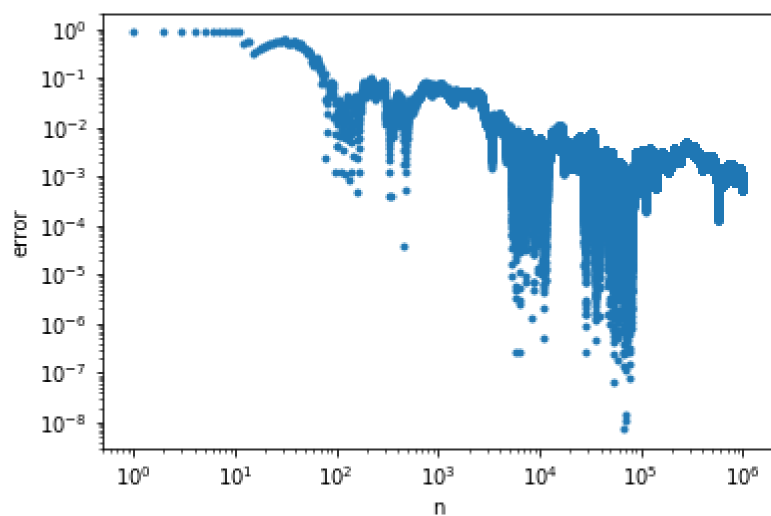
Out[24]: [<matplotlib.lines.Line2D at 0x11275f978>]



Which as expected gives an approximation for the number π, since the circle has radius 1. Let's plot the error of our approximation:

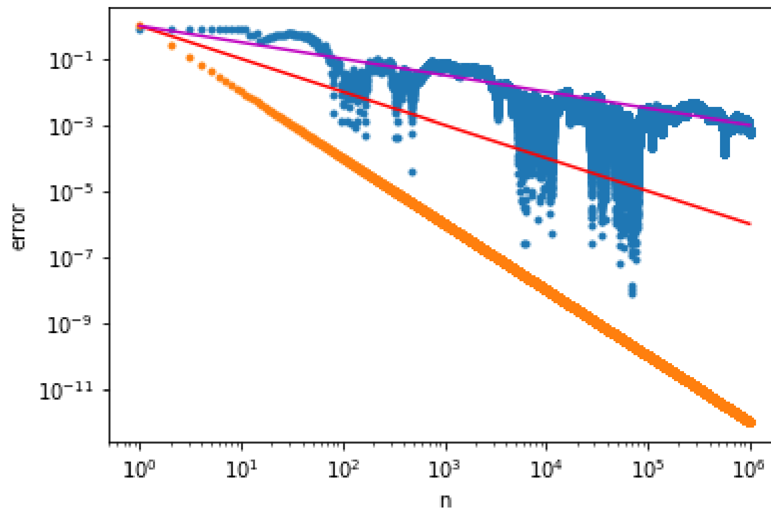
In [25]:
```
plt.loglog(N_samples, np.abs(A_approx - np.pi), '.')
plt.xlabel('n')
plt.ylabel('error')
```

Out[25]: Text(0, 0.5, 'error')



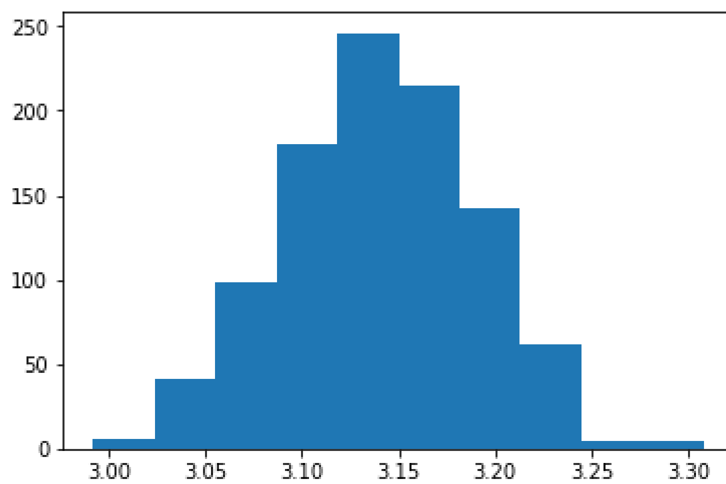
In [26]:
```
plt.loglog(N_samples, np.abs(A_approx - np.pi), '.')
plt.xlabel('n')
plt.ylabel('error')

plt.loglog(N_samples, 1/N_samples**2, '.')
plt.loglog(N_samples, 1/N_samples, 'r')
plt.loglog(N_samples, 1/np.sqrt(N_samples), 'm')
```

Out[26]: [<matplotlib.lines.Line2D at 0x112d00a58>]



In [27]:
```
N = 10**3
M = 1000

A_list = []

for i in range(M):
    samples = np.random.rand(2, N)
    xy = samples * 2 - 1.0 # scale sample points
    r = np.sqrt(xy[0, :]**2 + xy[1, :]**2)  # calculate radius
    incircle = (r <= 1)
    count_incircle = incircle.sum()
    A_list.append( (2*2) * (count_incircle)/N )

A_array = np.array(A_list)

plt.hist(A_list)
```

Out[27]:
```
(array([  6.,  41.,  98., 180., 246., 215., 142.,  62.,   5.,   5.]),
 array([2.992 , 3.0236, 3.0552, 3.0868, 3.1184, 3.15  , 3.1816, 3.2132,
        3.2448, 3.2764, 3.308 ]),
 <a list of 10 Patch objects>)
```



Approximating probabilities of a Poker game

What is the probability of winning a hand of Texas Holdem for a given starting hand?

Assumptions for this example:

- You are playing against only one player (the opponent)
- Both players stay until the end of a game (all 5 dealer cards), so players do not fold.

Monte Carlo simulation: for a given "starting hand" that we wish to obtain the winning probability, generate a set of N games and use the Texas Holdem rules to decide who wins (starting hand, opponent or there is a tie).

A game is a set of 7 random cards: 2 cards for opponent + 5 cards for the dealer.

For example, suppose the starting hand is 5 of clubs and 4 of diamonds. You perform N=1000 games, and counts 350 wins, 590 losses and 60 ties. Your numerical experiment estimated a probability of winning of 35%.

This is your first MP!