CA3_Asset_Pricing_instructor

October 1, 2019

```
In [1]: import numpy as np
        import matplotlib.pyplot as plt
        %matplotlib inline
```

1 Asset Pricing

Suppose that we are interested in investing in a specific stock. We may want to try and predict what the future price might be with some probability in order for us to determine whether or not we should invest.

The simplest way to model the movement of the price of an asset in a market with no moving forces is to assume that the price changes with some random magnitude and direction (the random walk theory)

We will implement this "random walk" by first simulating the roll of a dice.

1.0.1 Write a function called `dice` that will roll an integer number from 1 to 6.

```
In [2]: #clear
        dice = lambda: np.random.randint(1,7)
```

The function `dice` will now tell us by how much the price is changing after one time step. Now, we need to know whether the price is increasing or decreasing.

How can we determine whether to increase or decrease the price? By doing a coin flip!

1.0.2 Write a function called `flip` that will randomly choose between -1 (decreasing price) or +1 (increasing price).

Hint: Using `numpy.random.choice might` be helpful here.

```
In [3]: #clear
        flip = lambda: np.random.choice([1, -1])
```

By combining these two functions, we are able to obtain the price change at a given time.

Here we will assume that a coin flip combined with a dice roll gives the price change for a given day.

1.0.3 1) Numerical experiment to simulate the asset price variation for a period of 10 days

Create the numpy array `simulation10` to store the random asset price variation for each day using the functions `dice` and `flip`

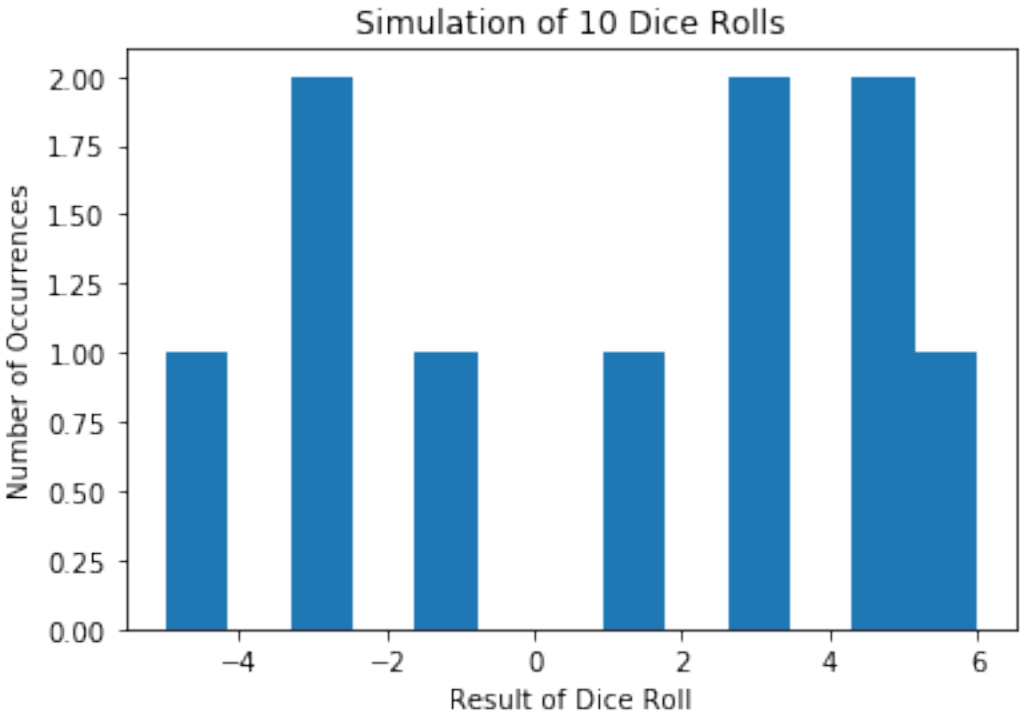
```
In [4]: N = 10 # number of days
```

```
In [5]: #clear
        # perform 10 steps of our simulation
        simulation10 = np.array([flip() * dice() for i in range(N)])
```

Use `plt.hist` to plot the histogram of the distribution of asset price variation (i.e. your `simulation10` array). We basically want to see how many times our simulation gives us -1, or -2, and so on...

```
In [6]: plt.hist(simulation10, 13)
        plt.title('Simulation of 10 Dice Rolls')
        plt.xlabel('Result of Dice Roll')
        plt.ylabel('Number of Occurrences')
```

```
Out[6]: Text(0, 0.5, 'Number of Occurrences')
```



What would you expect the histogram to look like if we repeated the numerical experiment above for a significantly larger amount of time steps (more days)?

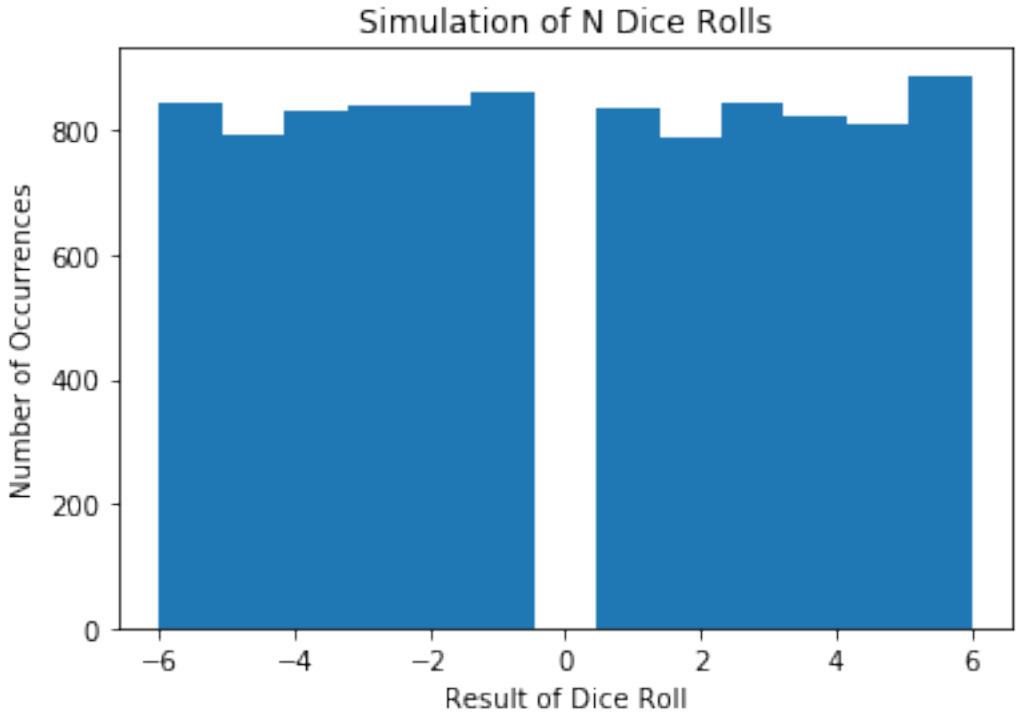
Can you tell what type of distribution we are sampling from in this simulation?

Try to reflect about the above questions before repeating the numerical experiment for N = 10000 days

```
In [7]: N = 10000

In [8]: #clear
        simulation = np.array([flip() * dice() for i in range(N)])
        plt.hist(simulation,13)
        plt.title('Simulation of N Dice Rolls')
        plt.xlabel('Result of Dice Roll')
        plt.ylabel('Number of Occurrences')

Out[8]: Text(0, 0.5, 'Number of Occurrences')
```



The above numerical experiment simulates how much the price changes each day, but does **NOT** calculate the actual asset price after each day.

How can we use the above results to find the asset price after each day?

We can accomplish this by performing a prefix sum. If we have an array v, a prefix sum can be written as

$$v[i] = \sum_{j=0}^{i} v[j],$$

where we repeat this for every element in our array.

Fortunately for us, python has a built-in function that performs a prefix sum called `cumsum`.

1.0.4 2) Numerical experiment to determine the asset price for each day over a period of 10 days

Assume the initial price of the stock is 0. Store the cumulative sum in the array `price10`.

```
In [9]: #clear
        price10 = simulation10.cumsum()
```

Use `print(price10)` and `print(simulation10)` to take a look at your results. Do you get the expected results according to the expression aboce?

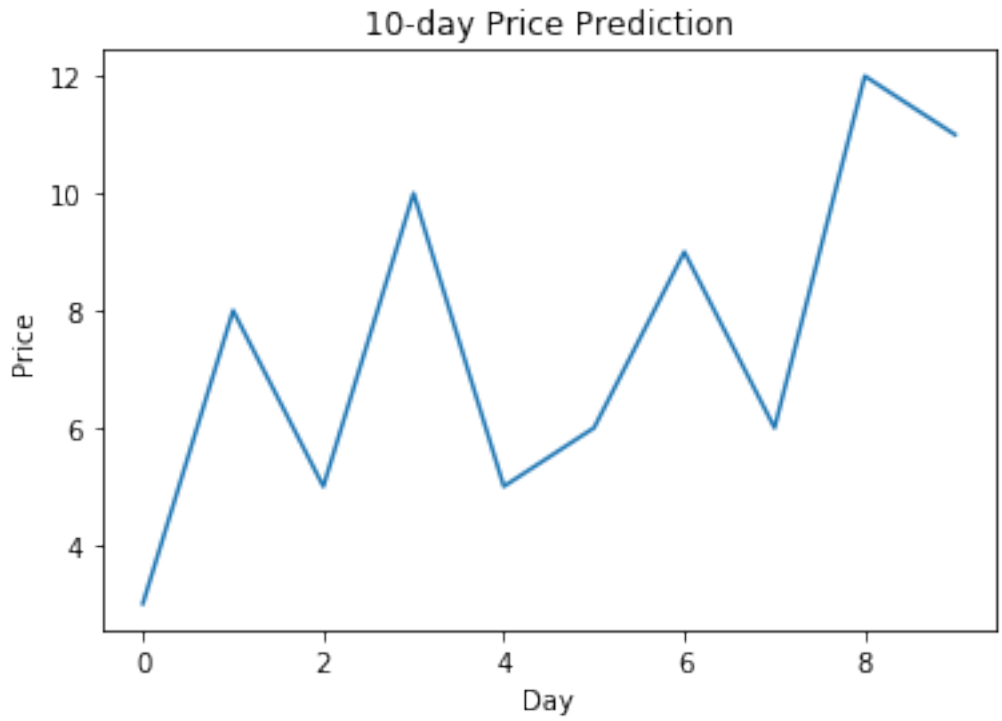
```
In [10]: #clear
         print(simulation10)
         print(price10)
```

```
[ 3  5 -3  5 -5  1  3 -3  6 -1]
[ 3  8  5 10  5  6  9  6 12 11]
```

Plot your results using `plt.plot(price10)`

```
In [11]: #plot prices
         plt.plot(price10)
         plt.title('10-day Price Prediction')
         plt.xlabel('Day')
         plt.ylabel('Price')
```

```
Out[11]: Text(0, 0.5, 'Price')
```



Does this plot resemble the short-term movement of the stock market? Let's repeat the numerical experiment above over a longer period of time

1.0.5 3) Numerical experiment to determine the asset price for each day over a period of 1000 days

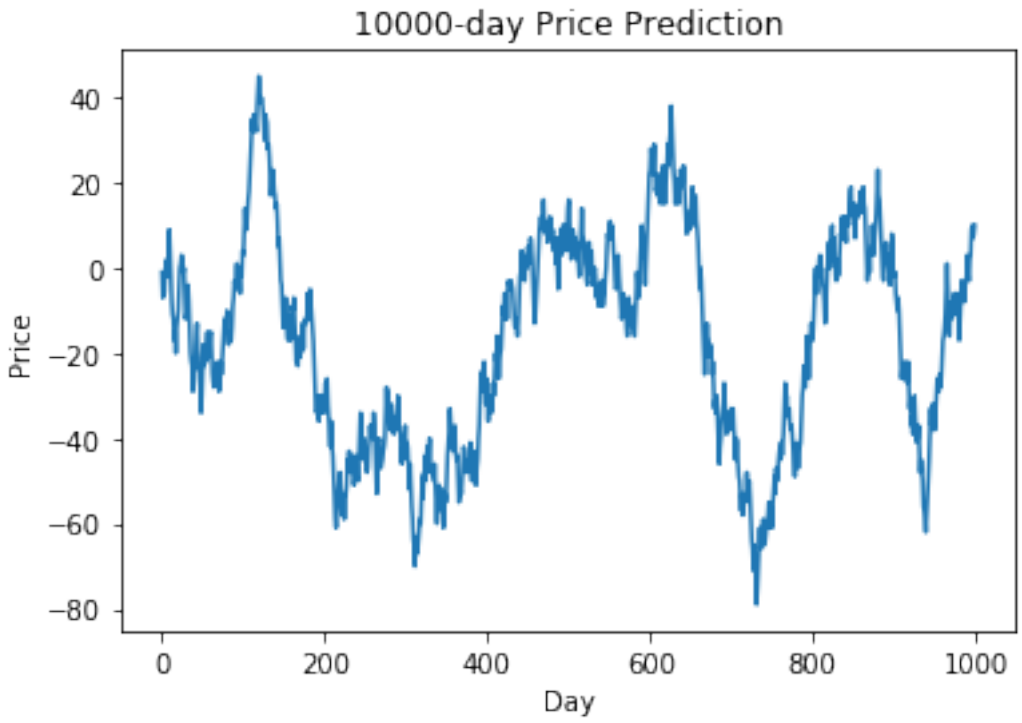
Assume the initial price of the stock is 0. Store the cumulative sum in the array `price` and plot your results using `plt.plot`

```
In [12]: #clear
         N = 1000

         # run same simulation but now with 100 time steps
         simulation = np.array([flip() * dice() for i in range(N)])

         price = simulation.cumsum()
         plt.plot(price)
         plt.title('10000-day Price Prediction')
         plt.xlabel('Day')
         plt.ylabel('Price')

Out[12]: Text(0, 0.5, 'Price')
```



Observations:

Performing one time step per day may not be enough to fully capture the randomness of the motion of the market. In practice, these N steps would really represent what the price might be in some shorter period of time (much less than a whole day).

Furthermore, performing a single numerical experiment will not give us a realistic expectation of what the price of the stock might be after a certain amount of time since the stock market with no moving forces consists of random movements.

Run the code snippet above several times (just do shift-enter again and again). What happens to the asset price after 1000 days?

1.0.6 4) Perform M=10 different numerical experiments, each one with N = 1000 days

For each numerical experiment, determine the array `price` using N = 1000 days. Make sure to store all the M=10 arrays `price` in the 2d array `prices_M`.

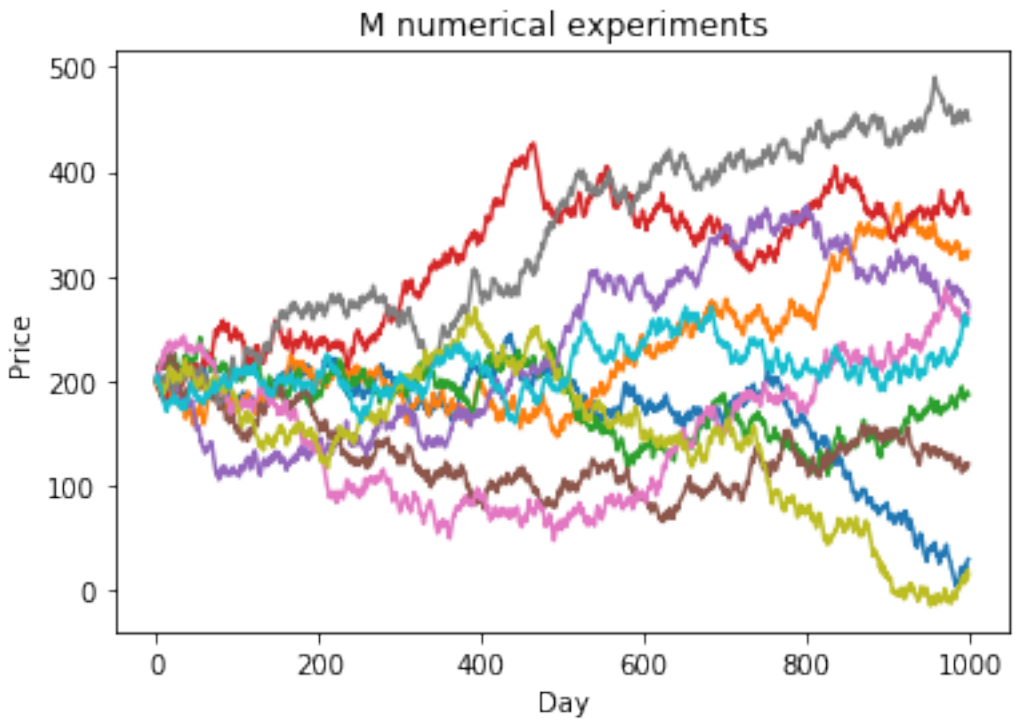
For this sequence of numerical experiments, assume that the initial asset price is `p0 = 200`

```
In [13]: N = 1000 # days
         M = 10   # number of numerical experiments
         p0 = 200 # initial asset price

In [14]: #clear
         price_M = []
         for i in range(M):
             simulation = np.array([flip() * dice() for j in range(N-1)])
             simulation = np.insert(simulation,0,p0)
             price = simulation.cumsum()
             price_M.append(price)
         price_M = np.array(price_M).T
```

Then you can plot your results using:

```
In [15]: plt.figure()
         plt.plot(price_M);
         plt.title ('M numerical experiments');
         plt.xlabel('Day');
         plt.ylabel('Price');
```



We now have a more insightful prediction as to what the price of a given stock might be in the future. Suppose we want to predict the asset price at day 1000. We can just take the last element of the numpy array `price`!

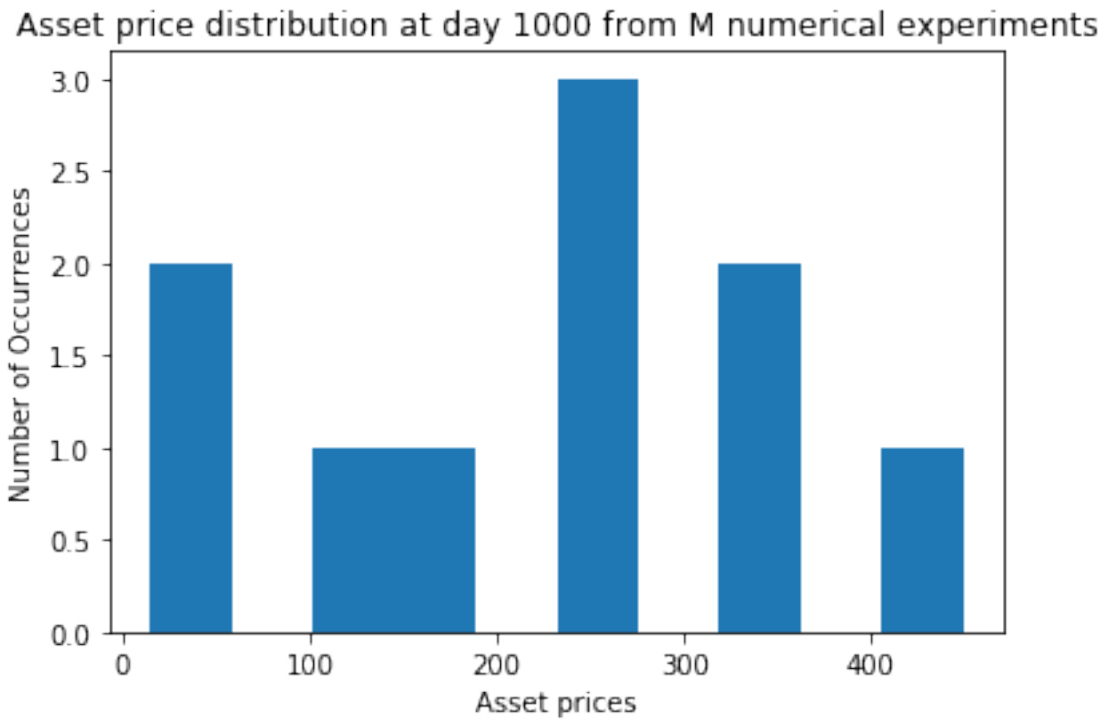
Create the variable `predicted_prices` to store the predicted asset prices for day 1000 for all the M=10 numerical experiments.

```
In [16]: #clear
         predicted_prices = price_M[-1,:]
```

Plot the histogram of the predicted price:

```
In [17]: plt.figure()
         plt.hist(predicted_prices);
         plt.title('Asset price distribution at day 1000 from M numerical experiments')
         plt.xlabel('Asset prices')
         plt.ylabel('Number of Occurrences')

Out[17]: Text(0, 0.5, 'Number of Occurrences')
```



Go back and change the number of numerical experiments. Set M = 1000 and run again. Better right?

You can calculate the mean of the distribution to get the "expected value" for the stock on day 1000. What do you get?

```
In [18]: predicted_prices.mean()
```

```
Out[18]: 228.3
```

There is one problem with our simple model. Our model does not incorporate any information about our specific stock other than the starting price. In order for us to get a more accurate model, we need to find a way incorporate the previous price of the stock.

1.1 Black-Scholes Model

We will now model stock price behavior using the Black-Scholes model, which employs a type of log-normal distribution to represent the growth of the stock price. Conceptually, this representation consists of two pieces:

1.1.1 1) Growth based on interest rate only

A simplified model based only on the compound interest rate r, would tell us that the stock price increases by e^r at every time increment, meaning $S_T = S_{T-1}e^r$. Extrapolating the compounded interest growth would imply that

$$S_T = S_t e^{r\Delta t}$$

where

- S_t price of the asset at time t
- S_T predicted price of the asset at time T
- r is the interest rate
- Δt is the time remaining $(T - t)$

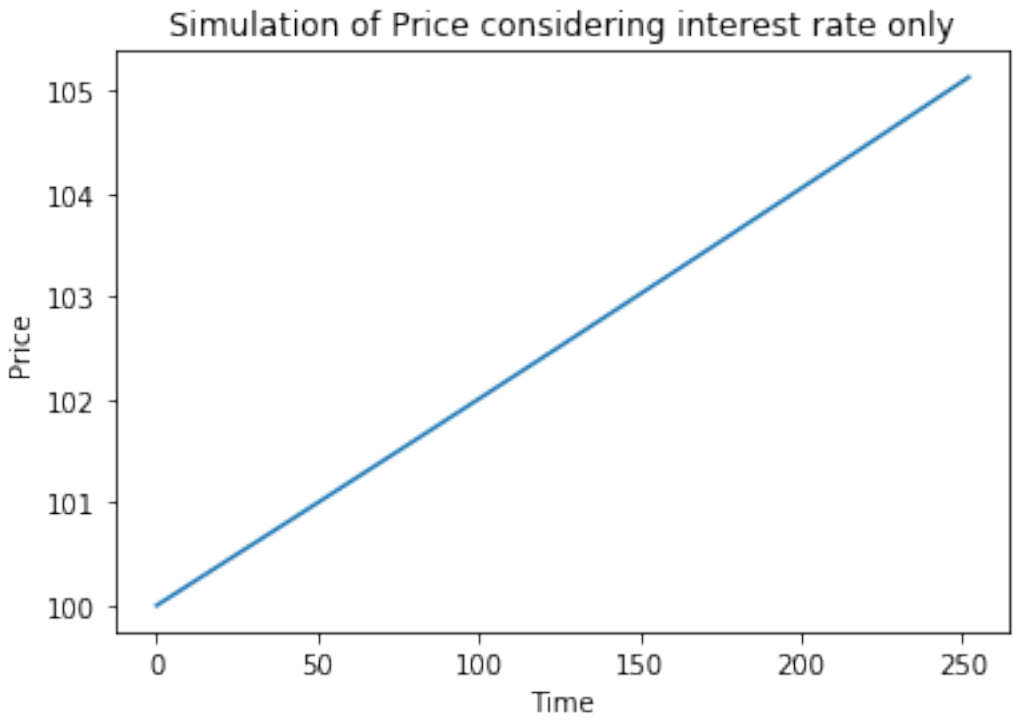
Daily price movements based on interest rate only Assume that at the initial time $t = 0$ the asset price is $S0 = 100$ and the interest rate is $r = 0.05$. Calculate the daily price movements according to the expression above for a period of 252 days (typical number of trading days in a year). Note that here the unit of t is days. Store your results in the array `price_interest_only`. Then plot your results using `plt.plot(price_interest_only)`

```
In [19]: #clear
         N = 252 # number of days
         S0 = 100
         r = 0.05
         deltaT = 1/N

         price_interest_only = np.array([S0])
         for i in range(N):
             st = price_interest_only[-1]
             price_interest_only = np.append(price_interest_only, st*np.exp(r*deltaT) )
```

```
In [20]: plt.plot(price_interest_only)
         plt.title('Simulation of Price considering interest rate only')
         plt.xlabel('Time')
         plt.ylabel('Price')

Out[20]: Text(0, 0.5, 'Price')
```



1.1.2 2) Add parameter to model volatility of the market

Stock prices evolve over time, with a magnitude dependent on their volatility. The Black Scholes model treats this evolution in terms of a random walk (a sequence of increments/decrements). To use the Black-Scholes model we assume:

- Some volatility or an annualized standard deviation of stock price. Call this σ
- We have a (risk-free) interest rate called r; and
- The price of the asset is geometric Brownian motion, or in other words the log of the random process is a normal distribution.

which leads to the following expression for the predicted asset price:

$$S_T = S_t e^{(r-\frac{\sigma^2}{2})\Delta T + \sigma\sqrt{\Delta T}\epsilon}$$

where

- σ is the volatility, or standard deviation on returns.
- ϵ is a random value sampled from the normal distribution $\mathcal{N}(0,1)$

Write a function `St_GBM` that will compute the price of an asset after a period ΔT

```
def St_GBM(St, r, sigma, deltat):
    ST = ... # Calculate this
    return ST
```

```
In [21]: #clear
         def St_GBM(St, r, sigma,deltat):
             epsilon = deltat**0.5*sigma*np.random.normal()
             S = St * np.exp( (r-sigma**2/2)*deltat + epsilon)
             return S
```

This model now gives us a more accurate way to predict the future price.

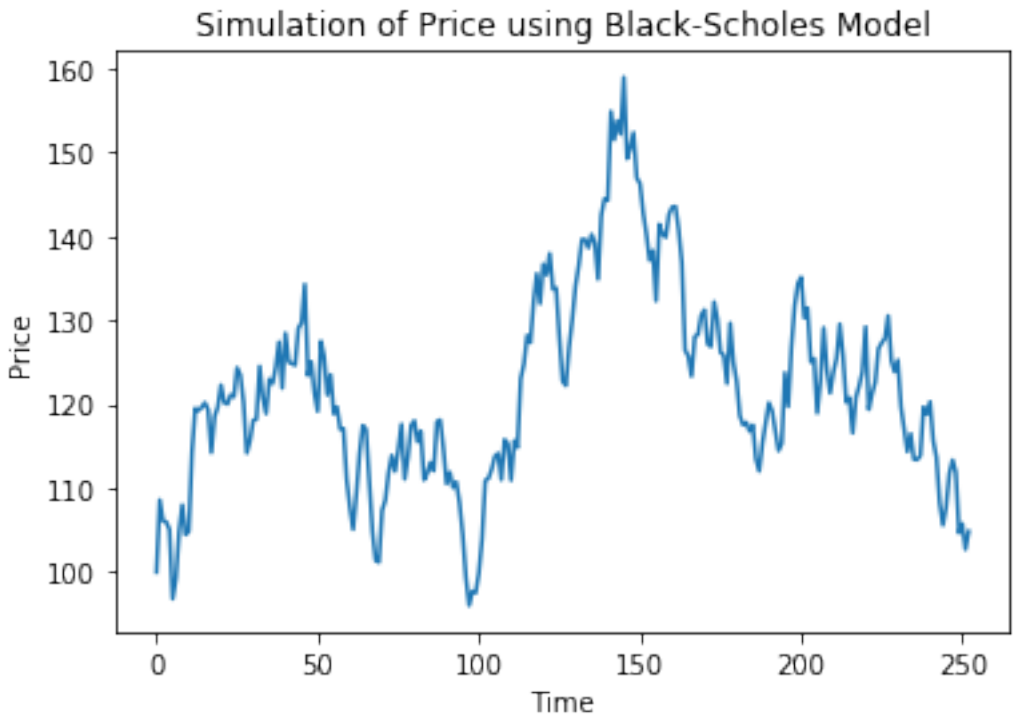
Daily price movements based on interest rate and volatility Assume that at the initial time $t = 0$ the asset price is $S0 = 100$, the interest rate is $r = 0.05$ and volatity is $\sigma = 0.1$. Calculate the daily price movements using `St_GBM` for a period of 252 days (typical number of trading days in a year). Note that here the unit of t is days. Store your results in the array `price`. Then plot your results using `plt.plot(price)`

```
In [22]: #clear
         N = 252 # number of days
         S0 = 100
         r = 0.05
         sigma = 0.5 #0.01
         deltaT = 1/N

         price = np.array([S0])
         for i in range(N):
             st = price[-1]
             price = np.append(price, St_GBM(price[-1],r,sigma,deltaT) )
```

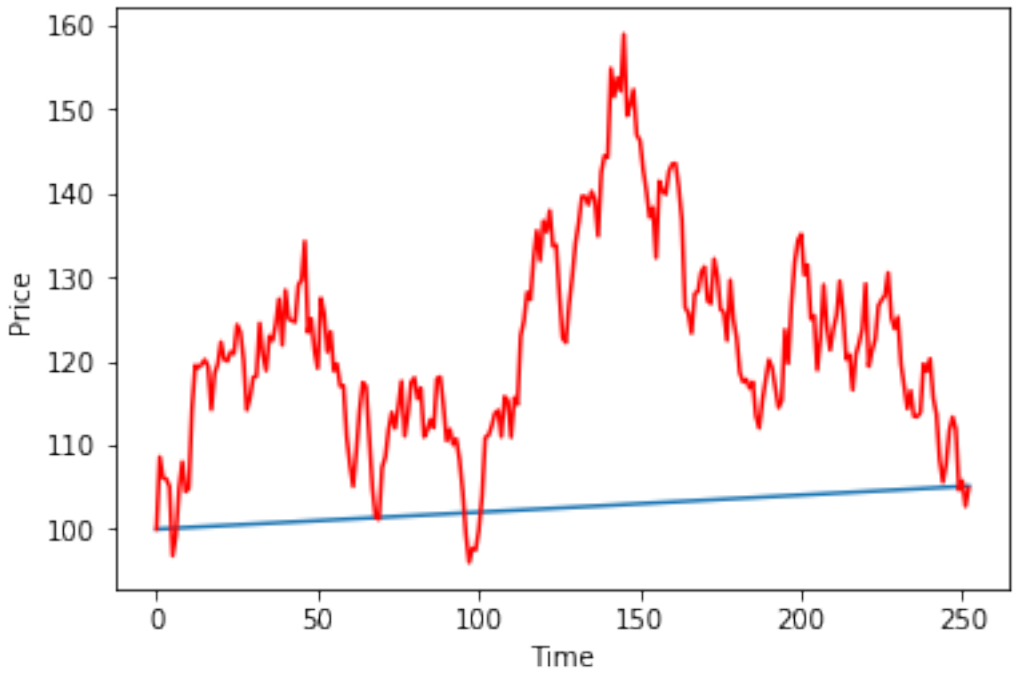
```
In [23]: plt.plot(price)
         plt.title('Simulation of Price using Black-Scholes Model')
         plt.xlabel('Time')
         plt.ylabel('Price')
```

```
Out[23]: Text(0, 0.5, 'Price')
```




```
In [24]: # Plot both models in the same figure
         plt.plot(price_interest_only)
         plt.plot(price,'r')
         plt.xlabel('Time')
         plt.ylabel('Price')

Out[24]: Text(0, 0.5, 'Price')
```



Unfortunately volatility is usually not this small.... Run the code snippet above to predict the price movement for a volatility $\sigma = 0.5$ So great – we have managed to successfully simulate a year's worth of future daily price data. Unfortunately this doesn't not provide insight into risk and return characteristics of the stock as we only have one randomly generated path. The likelyhood of the actual price evolving exactly as described in the above charts is pretty much zero. We should modify the aboce code to run multiple numerical experiments (or simulations).

1.1.3 Perform M=10 different numerical experiments, each one with N = 252 days

For each numerical experiment, determine the array `price` using N = 252 days. Make sure to store all the M=10 arrays `price` in the 2d array `prices_M`.

For this sequence of numerical experiments, assume that the initial asset price is `S0 = 100`.

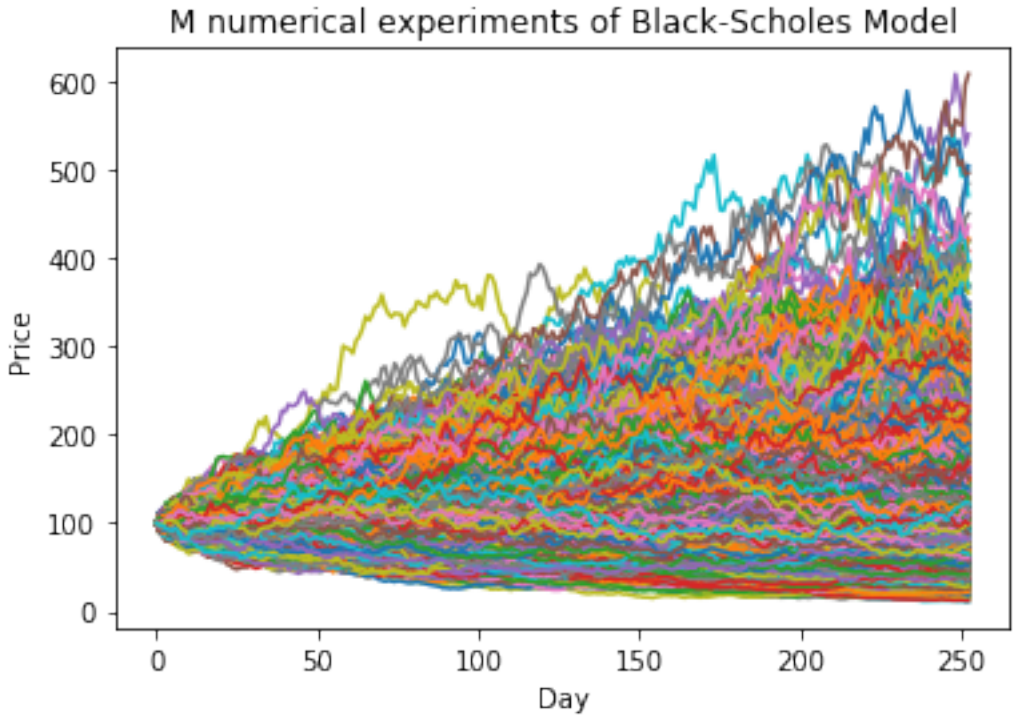
```
In [25]: N = 252  # days
         M = 10000  # number of numerical experiments
         S0 = 100 # initial asset price
         r = 0.05
         sigma = 0.5

In [26]: #clear
         # CODE SNIPPET A
         price_M = []
         for j in range(M):
             price = [S0]
```

```
        for i in range(N):
            price.append( St_GBM(price[-1],r,sigma,deltaT) )
        price_M.append(price)
    price_M = np.array(price_M).T
```

Then plot the result using:

```
In [27]: plt.figure()
         plt.plot(price_M);
         plt.title ('M numerical experiments of Black-Scholes Model');
         plt.xlabel('Day');
         plt.ylabel('Price');
```



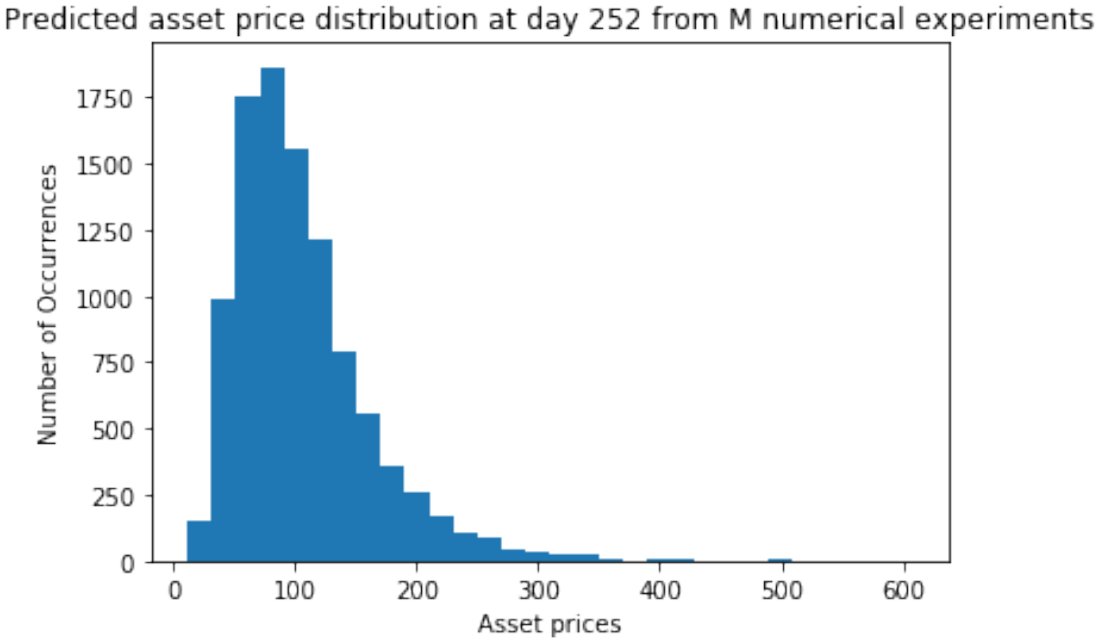
The spread of final prices is quite large! Let's take a further look at this spread. Create the variable `predicted_prices` to store the predicted asset prices for day 252 (last day) for all the M=10 numerical experiments.

```
In [28]: #clear
         predicted_prices = price_M[-1,:]
```

Plot the histogram of the predicted prices:

```
In [29]: plt.figure()
         plt.hist(predicted_prices,30);
         plt.title('Predicted asset price distribution at day 252 from M numerical experiments')
         plt.xlabel('Asset prices')
         plt.ylabel('Number of Occurrences')
```

```
Out[29]: Text(0, 0.5, 'Number of Occurrences')
```



Calculate the mean and standard deviation of the distribution for the stock on the last day. What do you get?

```
In [30]: #clear
         print( predicted_prices.mean(), predicted_prices.std() )

105.57162251849806 55.56064496837148
```

Congratulations! You now have a prediction for a future price for a given stock.

But wait ... Do you think you would get a similar prediction if you were to run the above code snippet again?

Can you do better?