**Monte Carlo Simulation and Density Functions**

Monte Carlo simulation (MCS) applies repeated random sampling (randomness) to obtain numerical results for deterministic problem solving. It is widely used in optimization, numerical integration, and risk-based decision making. Probability and cumulative density functions are statistical measures that apply probability distributions for random variables, and can be used in conjunction with MCS to solve deterministic problem:

1. stock simulations
2. What-If analysis
3. product demand simulation
4. randomness using probability and cumulative density functions

*Stock Simulations*

The 1st example is hypothetical and simple, but useful in demonstrating data randomization. It begins with a fictitious stock priced at 20 dollars. It then projects price out 200 days and plots.  
  
import matplotlib.pyplot as plt, numpy as np

from scipy import stats

def cum\_price(p, d, m, s):

data = []

for d in range(d):

prob = stats.norm.rvs(loc=m, scale=s)

price = (p \* prob)

data.append(price)

p = price

return data

if \_\_name\_\_ == "\_\_main\_\_":

stk\_price, days, mean, s = 20, 200, 1.001, 0.005

data = cum\_price(stk\_price, days, mean, s)

plt.plot(data, color='lime')

plt.ylabel('Price')

plt.xlabel('days')

plt.title('stock closing prices')

plt.show()  
  
Output:



The code begins by importing matplotlib, numpy and scipy libraries. It continues with function cum\_price(), which generates 200 normally distributed random numbers (one for each day) with norm\_rvs(). Data randomness is key. The main block creates the variables. Mean is set a bit over 1 and standard deviation (s) at a very small number to generate a slowly increasing stock price. Mean (mu) is the average change in value. Standard deviation is the variation or dispersion in the data. With s of 0.005, our data has very little variation. That is, the numbers in our data set are very close to each other. Remember that this is not a real scenario! The code continues by plotting results.  
  
The next example adds MCS into the mix with a while loop that iterates 100 times:

import matplotlib.pyplot as plt, numpy as np

from scipy import stats

def cum\_price(p, d, m, s):

data = []

for d in range(d):

prob = stats.norm.rvs(loc=m, scale=s)

price = (p \* prob)

data.append(price)

p = price

return data

if \_\_name\_\_ == "\_\_main\_\_":

stk\_price, days, mu, sigma = 20, 200, 1.001, 0.005

x = 0

while x < 100:

data = cum\_price(stk\_price, days, mu, sigma)

plt.plot(data)

x += 1

plt.ylabel('Price')

plt.xlabel('day')

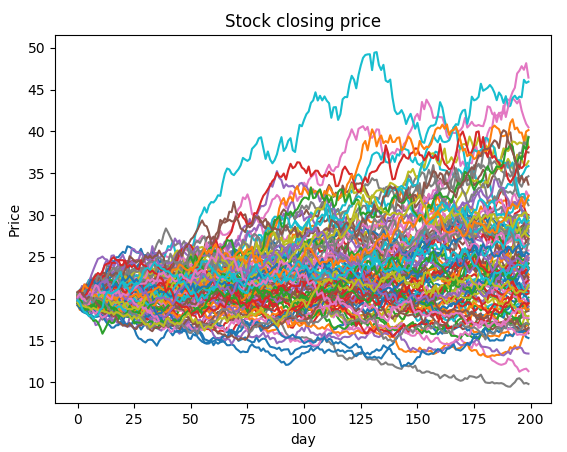
plt.title('Stock closing price')

plt.show()

Output:  
  


The while loop allows us to visualize 100 possible stock price outcomes over 200 days. Notice that mu (mean) and sigma (standard deviation) are used. This example demonstrates the power of MCS for decision making.  
  
*What-If analysis*  
  
What-If analysis changes values in an algorithm to see how they impact outcomes. Be sure to only change one variable at a time, otherwise you won’t know which caused the change. In the previous example, what if we change days to 500 while keeping all else constant (the same)? Plotting this change results in the following:  
  
  
  
Notice that the change in price is slower. Changing mu (mean) to 1.002 (don’t forget to change days back to 200) results in faster change (larger averages) as follows:



Changing sigma to 0.02 results in more variation as follows:  
  
  
  
*Product Demand Simulation*

A discrete probability is the probability of each discrete random value occurring in a sample space or population. A random variable assumes different values determined by chance. A discrete random variable can only assume a countable number of values. In contrast, a continuous random variable can assume an uncountable number of values in a line interval such as a normal distribution.

In the code example, demand for a fictitious product is predicted by four discrete probability outcomes – 10% that random variable is 10,000 units, 35% that random variable is 20,000 units, 30% that random variable is 40,000 units, and 25% that random variable is 60,000 units. Simply, 10% of the time demand is 10,000, 35% of the time demand is 20,000, 30% of the time demand is 40,000, and 25% of the time demand is 60,000. Discrete outcomes must total 100%. The code runs MCS on a production algorithm that determines profit for each discrete outcome, and plots the results.

import matplotlib.pyplot as plt, numpy as np

def demand():

p = np.random.uniform(0,1)

if p < 0.10:

return 10000

elif p >= 0.10 and p < 0.45:

return 20000

elif p >= 0.45 and p < 0.75:

return 40000

else:

return 60000

def production(demand, units, price, unit\_cost, disposal):

units\_sold = min(units, demand)

revenue = units\_sold \* price

total\_cost = units \* unit\_cost

units\_not\_sold = units - demand

if units\_not\_sold > 0:

disposal\_cost = disposal \* units\_not\_sold

else:

disposal\_cost = 0

profit = revenue - total\_cost - disposal\_cost

return profit

def mcs(x, n, units, price, unit\_cost, disposal):

profit = []

while x <= n:

d = demand()

v = production(d, units, price, unit\_cost, disposal)

profit.append(v)

x += 1

return profit

def max\_bar(ls):

tup = max(enumerate(ls))

return tup[0] - 1

if \_\_name\_\_ == "\_\_main\_\_":

units = [10000, 20000, 40000, 60000]

price, unit\_cost, disposal = 4, 1.5, 0.2

avg\_p = []

x, n = 1, 10000

profit\_10 = mcs(x, n, units[0], price, unit\_cost, disposal)

avg\_p.append(np.mean(profit\_10))

print ('Profit for {:,.0f}'.format(units[0]),

'units: ${:,.2f}'.format(np.mean(profit\_10)))

profit\_20 = mcs(x, n, units[1], price, unit\_cost, disposal)

avg\_p.append(np.mean(np.mean(profit\_20)))

print ('Profit for {:,.0f}'.format(units[1]),

'units: ${:,.2f}'.format(np.mean(profit\_20)))

profit\_40 = mcs(x, n, units[2], price, unit\_cost, disposal)

avg\_p.append(np.mean(profit\_40))

print ('Profit for {:,.0f}'.format(units[2]),

'units: ${:,.2f}'.format(np.mean(profit\_40)))

profit\_60 = mcs(x, n, units[3], price, unit\_cost, disposal)

avg\_p.append(np.mean(profit\_60))

print ('Profit for {:,.0f}'.format(units[3]),

'units: ${:,.2f}'.format(np.mean(profit\_60)))

labels = ['10000','20000','40000','60000']

pos = np.arange(len(labels))

width = 0.75 # set less than 1.0 for spaces between bins

plt.figure(2)

ax = plt.axes()

ax.set\_xticks(pos + (width / 2))

ax.set\_xticklabels(labels)

barlist = plt.bar(pos, avg\_p, width, color='aquamarine')

barlist[max\_bar(avg\_p)].set\_color('orchid')

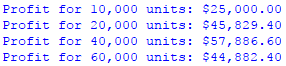
plt.ylabel('Profit')

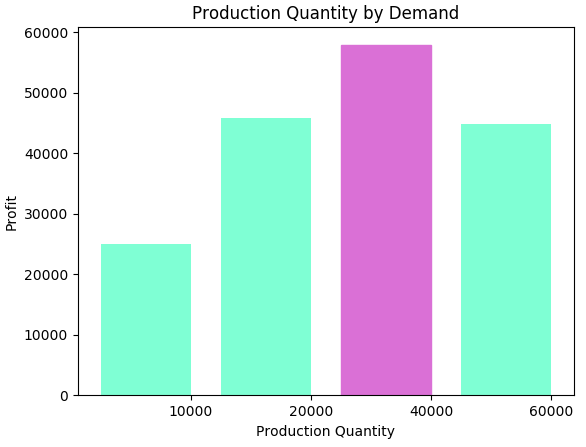
plt.xlabel('Production Quantity')

plt.title('Production Quantity by Demand')

plt.show()

Output:



  
  
The code begins by importing matplotlib and numpy libraries. It continues with four functions. Function demand() begins by randomly generating a uniformly distributed probability. It continues by returning one of the four discrete probability outcomes established by the problem we wish to solve. Function production() returns profit based on an algorithm that I devised. Keep in mind that any profit-base algorithm can be substitued, which illuminates the incredible flexibility of MCS. Function mcs() runs the simulation 10,000 times. Increasing the number of runs provides better prediction accuracy with costs being more computer processing resources and runtime. Function max\_bar() establishes the highest bar in the bar chart for better illumination. The main block begins by simulating profit for each discrete probability outcome, and printing and visualizing results. MCS predicts that production quantity of 40,000 units yields the highest profit.

Increasing the number of MCS simulations results in a more accurate prediction of reality because it is based on stochastic reasoning (data randomization). You can also substitute any discrete probability distribution based on your problem-solving needs with this code structure. As alluded to earlier, you can use any algorithm you wish to predict with MCS making it an incredibly flexible tool for data scientists.

We can further enhance accuracy by running a MCS on a MCS. The code example uses the same algorithm and process as before, but adds a MCS on the original MCS to get a more accurate prediction.

import matplotlib.pyplot as plt, numpy as np

def demand():

p = np.random.uniform(0,1)

if p < 0.10:

return 10000

elif p >= 0.10 and p < 0.45:

return 20000

elif p >= 0.45 and p < 0.75:

return 40000

else:

return 60000

def production(demand, units, price, unit\_cost, disposal):

units\_sold = min(units, demand)

revenue = units\_sold \* price

total\_cost = units \* unit\_cost

units\_not\_sold = units - demand

if units\_not\_sold > 0:

disposal\_cost = disposal \* units\_not\_sold

else:

disposal\_cost = 0

profit = revenue - total\_cost - disposal\_cost

return profit

def mcs(x, n, units, price, unit\_cost, disposal):

profit = []

while x <= n:

d = demand()

v = production(d, units, price, unit\_cost, disposal)

profit.append(v)

x += 1

return profit

def display(p):

print ('Profit for {:,.0f}'.format(units[1]),

'units: ${:,.2f}'.format(np.mean(p)))

if \_\_name\_\_ == "\_\_main\_\_":

units = [10000, 20000, 40000, 60000]

price, unit\_cost, disposal = 4, 1.5, 0.2

avg\_ls = []

x, n, y, z = 1, 10000, 1, 1000

while y <= z:

profit\_10 = mcs(x, n, units[0], price, unit\_cost, disposal)

profit\_20 = mcs(x, n, units[1], price, unit\_cost, disposal)

avg\_profit = np.mean(profit\_20)

profit\_40 = mcs(x, n, units[2], price, unit\_cost, disposal)

avg\_profit = np.mean(profit\_40)

profit\_60 = mcs(x, n, units[3], price, unit\_cost, disposal)

avg\_profit = np.mean(profit\_60)

avg\_ls.append({'p10':np.mean(profit\_10),

'p20':np.mean(profit\_20),

'p40':np.mean(profit\_40),

'p60':np.mean(profit\_60)})

y += 1

mcs\_p10, mcs\_p20, mcs\_p40, mcs\_p60 = [], [], [], []

for row in avg\_ls:

mcs\_p10.append(row['p10'])

mcs\_p20.append(row['p20'])

mcs\_p40.append(row['p40'])

mcs\_p60.append(row['p60'])

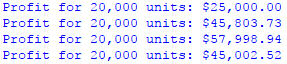
display(np.mean(mcs\_p10))

display(np.mean(mcs\_p20))

display(np.mean(mcs\_p40))

display(np.mean(mcs\_p60))

Output:



The code for this example is the same as the previous one, except for the MCS while loop (while y <= z). In this loop, profits are calculated as before using function mcs(), but each simulation result is appended to list avg\_ls. So, avg\_ls contains 1000 (z = 1000) simulation results of the original simulation results. Accuracy is increased, but more computer resources and runtime is required. Running 1000 simulations on the original MCS takes a bit over one minute, which is a lot of processing time!

*Randomness using Probability and Cumulative Density Functions*  
  
Randomness masquerades as reality (the natural world) in data science since the future cannot be predicted. That is, randomization is the way data scientists simulate reality. More data means better accuracy and prediction (more realism). It plays a key role in discrete event simulation and deterministic problem solving. Randomization is used in many fields such as statistics, MCS, cryptography, statistics, medicine, and science.

The density of a continuous random variable is its probability density function (PDF). PDF is the probability that a random variable has the value x, where x is a point within the interval of a sample. This probability is determined by the integral of the random variable’s PDF over the range (interval) of the sample. That is, the probability is given by the area under the density function, but above the horizontal axis and between the lowest and highest values of range. An integral (integration) is a mathematical object that can be interpreted as an area under a normal distribution curve. A cumulative distribution function (CDF) is the probability that a random variable has a value less than or equal to x. That is, CDF accumulates all of the probabilities less than or equal to x. The percent point function (PPF) is the inverse of the CDF. It is commonly referred to as the inverse cumulative distribution function (ICDF). ICDF is very useful in data science because it is the actual value associated with an area under the PDF. Please refer to <http://www.itl.nist.gov/div898/handbook/eda/section3/eda362.htm> for an excellent explanation of density functions.

As stated earlier, a probability is determined by the integral of the random variable’s PDF over the interval of a sample. That is, integrals are used to determine the probability of some random variable falling within a certain range (sample). In calculus, the integral represents a class of functions (the antiderivative) whose derivative is the integrand. The integral symbol represents integration, while an integrand is the function being integrated in either a definite or indefinite integral. The fundamental theorem of calculus relates the evaluation of definitive integrals to indefinite integrals. The only reason I include this information here is to emphasize the importance of calculus to data science. Another aspect of calculus important to data science is presented later in chapter 3 (gradient descent).

Although theoretical explanations are invaluable, they may not be intuitive. A great way to better understand these concepts is to look at an example.

In the code example, two dimensional (2D) charts are created for PDF, CDF, and ICDF (PPF). The idea of a colormap is included in the example. A colormap is a lookup table specifying the colors to be used in rendering palettized image. A palettized image is one that is efficiently encoded by mapping its pixels to a palette containing only those colors that are actually present in the image. The matplotlib library includes a myriad of colormaps. Please refer to <https://matplotlib.org/examples/color/colormaps_reference.html> for available colormaps.

import matplotlib.pyplot as plt

from scipy.stats import norm

import numpy as np

if \_\_name\_\_ == '\_\_main\_\_':

x = np.linspace(norm.ppf(0.01), norm.ppf(0.99), num=1000)

y1 = norm.pdf(x)

plt.figure('PDF')

plt.xlim(x.min()-.1, x.max()+0.1)

plt.ylim(y1.min(), y1.max()+0.01)

plt.xlabel('x')

plt.ylabel('Probability Density')

plt.title('Normal PDF')

plt.scatter(x, y1, c=x, cmap='jet')

plt.fill\_between(x, y1, color='thistle')

plt.show()

plt.close('PDF')

plt.figure('CDF')

plt.xlabel('x')

plt.ylabel('Probability')

plt.title('Normal CDF')

y2 = norm.cdf(x)

plt.scatter(x, y2, c=x, cmap='jet')

plt.show()

plt.close('CDF')

plt.figure('ICDF')

plt.xlabel('Probability')

plt.ylabel('x')

plt.title('Normal ICDF (PPF)')

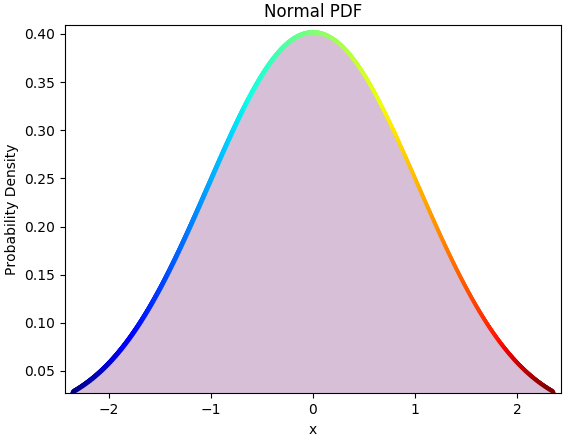
y3 = norm.ppf(x)

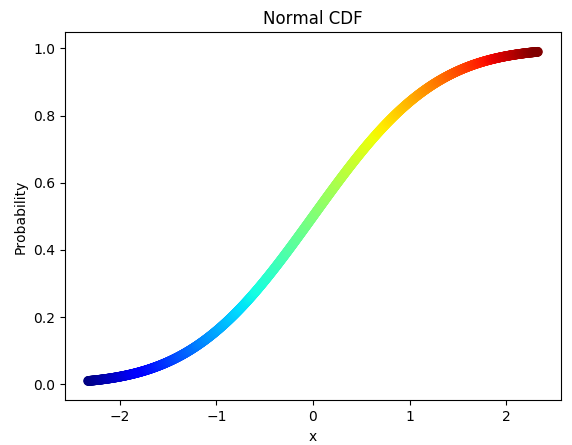
plt.scatter(x, y3, c=x, cmap='jet')

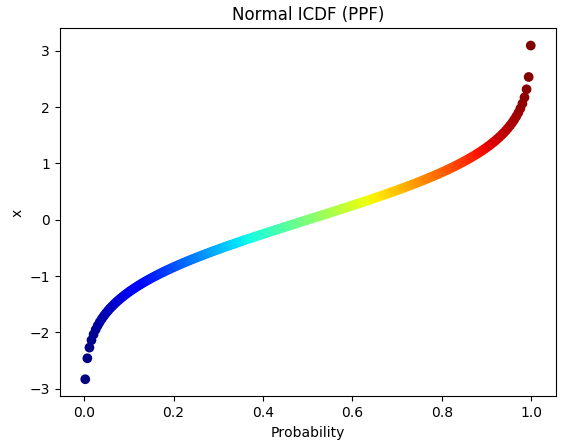
plt.show()

plt.close('ICDF')

Output:







The code begins by importing three libraries – matplotlib, scipy, and numpy. The main block begins by creating a sequence of 1000 x values between 0.01 and 0.99 (because probabilities must fall between 0 and 1). Next, a sequence of PDF y values is created based on the x values. The code continues by plotting the resultant PDF. Next, a sequence of CDF and ICDF values are created and plotted. From the visualization, it is easier to see that the PDF represents all of the possible x values (probabilities) that exist under the normal distribution. It is also easier to visualize the CDF because it represents the accumulation of all the possible probabilities. Finally, the ICDF is easier to understand through visualization because the x-axis represents probabilities, while the y-axis represents the actual value associated with those probabilities.

Let’s apply ICDF. Suppose you are a data scientist at Apple and your boss asks you to determine Apple iPhone 8 failure rates so she can develop a mockup presentation for her superiors. For this hypothetical example, your boss expects four calculations: time it takes 5% of phones to fail, time interval (range) where 95% of phones fail, time where 5% of phones survive (don’t fail), and time interval where 95% of phone survive. In all cases, report time in hours. From data exploration, you ascertain average (mu) failure time is 1000 hours and standard deviation (sigma) is 300 hours.

The code example calculates ICDF for the four scenarios and displays the results in an easy to understand format for your boss:

from scipy.stats import norm

import numpy as np

def np\_rstrip(v):

return np.char.rstrip(v.astype(str), '.0')

def transform(t):

one, two = round(t[0]), round(t[1])

return (np\_rstrip(one), np\_rstrip(two))

if \_\_name\_\_ == "\_\_main\_\_":

mu, sigma = 1000, 300

print ('Expected failure rates:')

fail = np\_rstrip(round(norm.ppf(0.05, loc=mu, scale=sigma)))

print ('5% fail within', fail, 'hours')

fail\_range = norm.interval(0.95, loc=mu, scale=sigma)

lo, hi = transform(fail\_range)

print ('95% fail between', lo, 'and', hi, end=' ')

print ('hours of usage')

print ('\nExpected survival rates:')

last = np\_rstrip(round(norm.ppf(0.95, loc=mu, scale=sigma)))

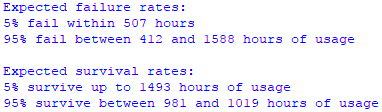
print ('5% survive up to', last, 'hours of usage')

last\_range = norm.interval(0.05, loc=mu, scale=sigma)

lo, hi = transform(last\_range)

print ('95% survive between', lo, 'and', hi, 'hours of usage')

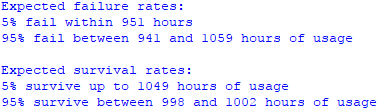
Output:



The code example begins by importing scipy and numpy libraries. It continues with two functions. Function np\_rstrip() converts numpy float to string and removes extraneous characters. Function transform() rounds and returns a tuple. Both are just used to round numbers to no decimal places to make it user-friendly for your fictitious boss. The main block begins by initializing mu and sigma to 1000 (failures) and 300 (variates). That is, on average, our smartphones fail within 1000 hours, and failures vary between 700 and 1300 hours. Next, find the ICDF value for a 5% failure rate and an interval where 95% fail with norm.ppf(). So, 5% of all phones are expected to fail within 507 hours, while 95% fail between 412 and 1588 hours of usage. Next, find the ICDF value for a 5% survival rate and an interval where 95% survive. So, 5% of all phones survive up to 1493 hours, while 95% survive between 981 and 1019 hours of usage.

Simply, ICDF allows you to work backwards from a known probability to find an x value! Please refer to <http://support.minitab.com/en-us/minitab-express/1/help-and-how-to/basic-statistics/probability-distributions/supporting-topics/basics/using-the-inverse-cumulative-distribution-function-icdf/#what-is-an-inverse-cumulative-distribution-function-icdf> for more information.

Let’s try What-if analysis. What if we reduce error rate (sigma) from 300 to 30?



Now, 5% of all phones are expected to fail within 951 hours, while 95% fail between 941 and 1059 hours of usage. And, 5% of all phones survive up to 1049 hours, while 95% survive between 998 and 1002 hours of usage. What does this mean? Less variation (error) shows that values are much closer to the average for both failure and survival rates. This makes sense because variation is calculated from a mean of 1000.

Let’s shift to a simulation example. Suppose your boss asks you to find the optimal monthly order quantity for a type of car given that demand is normally distributed (it must because PDF is based on this assumption), average demand (mu) is 200, and variation (sigma) is 30. Each car costs $25,000, sells for $45,000, and half of the cars not sold at full price can be sold for $30,000. Like other MCS experiments, you can modify the profit algorithm to enhance realism. By suppliers, you are limited to order quantities of 160, 180, 200, 220, 240, 260 or 280.

MCS is used to find the profit for each order based on the information provided. Demand is generated randomly for each iteration of the simulation. Profit calculations by order are automated by running MCS for each order.

import numpy as np

import matplotlib.pyplot as plt

def str\_int(s):

val = "%.2f" % profit

return float(val)

if \_\_name\_\_ == "\_\_main\_\_":

orders = [180, 200, 220, 240, 260, 280, 300]

mu, sigma, n = 200, 30, 10000

cost, price, discount = 25000, 45000, 30000

profit\_ls = []

for order in orders:

x = 1

profit\_val = []

inv\_cost = order \* cost

while x <= n:

demand = round(np.random.normal(mu, sigma))

if demand < order:

diff = order - demand

if diff > 0:

damt = round(abs(diff) / 2) \* discount

profit = (demand \* price) - inv\_cost + damt

else:

profit = (order \* price) - inv\_cost

else:

profit = (order \* price) - inv\_cost

profit = str\_int(profit)

profit\_val.append(profit)

x += 1

avg\_profit = np.mean(profit\_val)

profit\_ls.append(avg\_profit)

print ('${0:,.2f}'.format(avg\_profit), '(profit)',

'for order:', order)

max\_profit = max(profit\_ls)

profit\_np = np.array(profit\_ls)

max\_ind = np.where(profit\_np == profit\_np.max())

print ('\nMaximum profit', '${0:,.2f}'.format(max\_profit),

'for order', orders[int(max\_ind[0])])

barlist = plt.bar(orders, profit\_ls, width=15, color='thistle')

barlist[int(max\_ind[0])].set\_color('lime')

plt.title('Profits by Order Quantity')

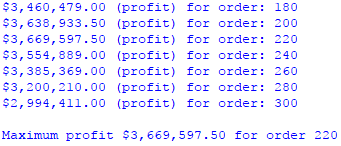
plt.xlabel('orders')

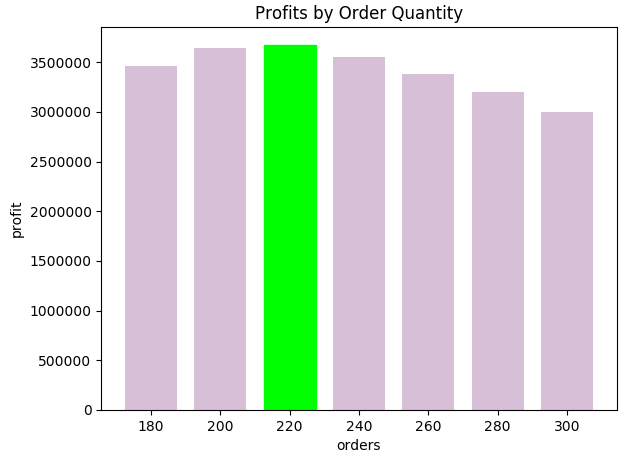
plt.ylabel('profit')

plt.tight\_layout()

plt.show()

Output:





The code begins by importing numpy and matplotlib. It continues with a function (str\_int()) that converts a string to float. The main block begins by initializing orders, mu, sigma, n, cost, price, discount, and list of profits by order. It continues by looping through each order quantity and running MCS with 10,000 iterations. A randomly generated demand probability is used to calculate profit for each iteration of the simulation. The technique for calculating profit is pretty simple, but you can substitute your own algorithm. You can also modify any of the given information based on your own data. After calculating profit for each order through MCS, the code continues by finding the order quantity with the highest profit. Finally, the code generates a bar chart to illuminate results though visualization.

The final code example creates a PDF visualization:

import matplotlib.pyplot as plt, numpy as np

from scipy.stats import norm

if \_\_name\_\_ == '\_\_main\_\_':

n = 100

x = np.linspace(norm.ppf(0.01), norm.ppf(0.99), num=n)

y = norm.pdf(x)

dic = {}

for i, row in enumerate(y):

dic[x[i]] = [np.random.uniform(0, row) for \_ in range(n)]

xs = []

ys = []

for key, vals in dic.items():

for y in vals:

xs.append(key)

ys.append(y)

plt.xlim(min(xs), max(xs))

plt.ylim(0, max(ys)+0.02)

plt.title('Normal PDF')

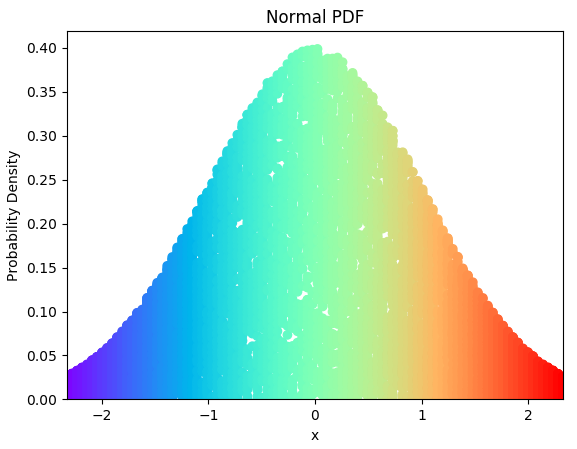
plt.xlabel('x')

plt.ylabel('Probability Density')

plt.scatter(xs, ys, c=xs, cmap='rainbow')

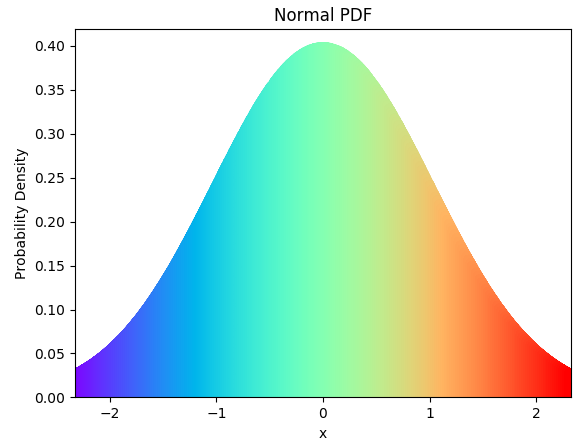
plt.show()

Output:



The code begins by importing matplotlib, numpy, and scipy libraries. The main block begins by initializing the number of points you wish to plot, PDF x and y values, and a dictionary. To plot all PDF probabilities, a set of randomly generated values for each point on the x-axis is created. To accomplish this task, the code assigns 100 (n=100) values to x from 0.01 to 0.99. It continues by assigning 100 PDF values to y. Next, a dictionary element is populated by a (key, value) pair consisting of each x value as key and a list of 100 (n=100) randomly generated numbers between 0 and pdf(x) as value associated with x. Although the code creating the dictionary is simple, please think carefully about what is happening because it is pretty abstract. The code continues by building (x, y) pairs from the dictionary. The result is 10,000 (100 X 100) (x, y) pairs, where each 100 x values has 100 associated y values.

To smooth out the visualization increase n to 1000 (n=100) at the beginning of the main block:



By increasing n to 1000, 1,000,000 (1000 X 1000) (x, y) pairs are plotted!