STAT 177, CLASS 9

Richard Waterman

July 2020

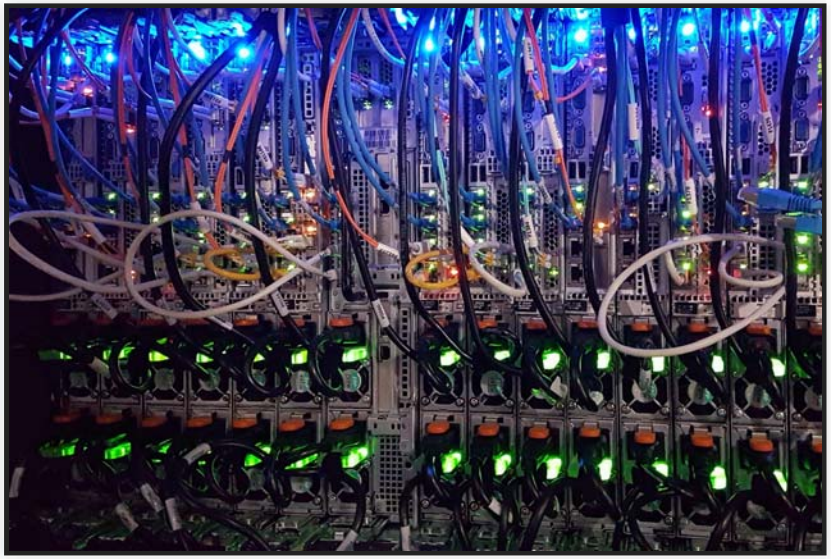
OBJECTIVES

OBJECTIVES

- An overview of scientific computing and numpy:
  - Examples from science.
- Vectors, matrices and arrays.
- Images as numbers.
- Monte Carlo simulation.
- Optimization algorithms.

SCIENTIFIC COMPUTING

SCIENTIFIC COMPUTING



SCIENTIFIC COMPUTING

- Computational methods for solving real world science problems. Among many examples are:
  - Biology: genomics (an interdisciplinary field of biology focusing on the structure, function, evolution, mapping, and editing of genomes).
  - Physics: The High-Luminosity Large Hadron Collider.

> *The next-generation of HPC technology offers great promise for supporting scientific research. Exascale supercomputers – machines capable of performing a quintillion, or a billion billion, calculations per second – are expected to become a reality in the next few years. This change in the power of HPC technology, coupled with growing use of machine learning , will be vital in ensuring the success of big science projects …*

SCIENTIFIC COMPUTING (CTD.)

- Chemistry: protein folding and pharmaceutical development.
- Engineering: computer vision and autonomous vehicles.
- Astronomy: wide scale sky surveys.
- What these areas have in common are massive data sets and the need for huge computational resources.
- We will also look at some more Monte Carlo simulation to solve probabilistic modeling problems.

CONFUSED VEHICLES

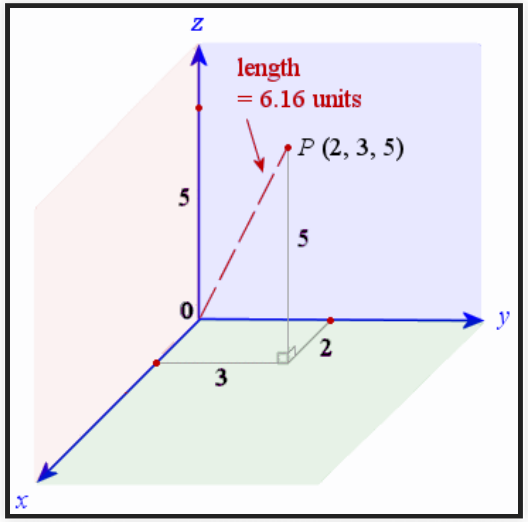
NUMPY

NUMPY

- We will use extremely popular ‘numpy’ library.
- numpy implements advanced mathematical operations on potentially very large data sets more efficiently than could be done with the native Python data structures.
- In particular it supports *arrays* , as the core data structure.
- An array will be more restrictive than a pandas data frame, but consequently can be processed more efficiently.

ONE DIMENSIONAL ARRAYS

- A one dimensional array is called a ‘vector’.
- An example of a vector is the position of a point in 3-space.
- There will be x, y and z components to the vector. Below is a picture of the point (2, 3, 5) in 3-space.



OPERATIONS ON 1-D ARRAYS

- A key operation for vectors is the "dot product".
- If we have two vectors x and y, where $x = \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix}$ and $y = \begin{pmatrix} y_1 \\ y_2 \\ y_3 \end{pmatrix}$, then their dot product, written as $x \cdot y$, is: $x_1y_1 + x_2y_2 + x_3y_3$.
- For an example let's say $x = \begin{pmatrix} 2 \\ 3 \\ 5 \end{pmatrix}$ and $y = \begin{pmatrix} 3 \\ 2 \\ 4 \end{pmatrix}$, then their dot product is $2 * 3 + 3 * 2 + 5 * 4 = 6 + 6 + 20 = 32$.
- In numpy these vectors would be called **arrays**.

IMPLEMENTING THE DOT PRODUCT WITH NUMPY

```
import numpy as np

x = np.array([2,3,5])
y = np.array([3,2,4])

np.dot(x,y) # The dot product between two vectors.
```

```
32
```

VECTORIZED OPERATIONS

- One of the key features of numpy is the the ‘vectorization’ of operations on the arrays.
- This means that we can apply functions to the array as a whole, and not have to write for loops to iterate over the arrays.
- For example, if we have two vectors, with the same number of elements we can do elementwise addition simple by using the ‘+’ operator.

```
print(x + y) # Addition element wise
print(x * y) # Multiplication elementwise
```

```
[5 5 9]
[ 6  6 20]
```

ADDING TWO LISTS?

- Look what happens if we had created two lists in Python and tried to add them:

```
x_1 = [2,3,5]
y_1 = [3,2,4]
x_1 + y_1 # Here, Python concatenated the two lists!
```

```
[2, 3, 5, 3, 2, 4]
```

NON-CONFORMING DIMENSIONS

- If the vectors do not have the same length and we try and add them, then:

```
a = np.array([2,3,5])
b = np.array([3,2,4,7,2,8])
print(a + b)
```

```
---------------------------------------------------------------------------

ValueError                                Traceback (most recent call last)

<ipython-input-4-07ca8b96321e> in <module>
      1 a = np.array([2,3,5])
      2 b = np.array([3,2,4,7,2,8])
----> 3 print(a + b)


ValueError: operands could not be broadcast together with shapes (3,) (6,)
```

THE DOT PRODUCT OF A VECTOR WITH ITSELF

- For the special case of a vector with three elements, the dot product $x \cdot x$ is $x_1^2 + x_2^2 + x_3^2$.
- That is, the sum of the squares of the individual elements.
- For the vector $x = \begin{pmatrix} 2 \\ 3 \\ 5 \end{pmatrix}$ this is $2^2 + 3^2 + 5^2 = 38$.
- The *length* of the vector is the square root of the dot product, here, $\sqrt{38} = 6.164$.
- This is written as $\|x\|$.

```
sq_length_x = np.dot(x,x) # The dot product command.
print("The length of the vector is", np.sqrt(sq_length_x)) # numpy has its own square root.
```

```
The length of the vector is 6.164414002968976
```

THE NORMALIZED DOT PRODUCT

- We will calculate the quantity:

$$\frac{x \cdot y}{\|x\|\|y\|}.$$

- This is in fact the cosine of the angle between the two vectors.

```
length_x = np.sqrt(np.dot(x,x))
length_y = np.sqrt(np.dot(y,y))

result = np.dot(x,y)/(length_x * length_y)
print(result)
```

```
0.9639603730060448
```

SPEED TEST

- We now create some very large vectors, and see how long numpy takes to calculate their inner product, as compared to a 'for loop'.

```
np.random.seed(0)
x1 = np.random.randn(10000000) # A long random vector
y1 = np.random.randn(10000000) # A long random vector

x2 = list(x1) # The same numbers as above, but now in a list.
y2 = list(y1)

print(type(x1)) # Confirming that we have two different data structures.
print(type(x2))
```

```
<class 'numpy.ndarray'>
<class 'list'>
```

TIME THE NUMPY DOT PRODUCT

- The %%timit in the cell is known as a “magic” and times how long the cell takes to run.

```
%%timeit
dot_p = np.dot(x1,y1)
```

```
8.69 ms ± 611 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

```
dot_p = np.dot(x1,y1)
```

TIMING A 'FOR LOOP'

```
%%timeit
counter = 0
for i in range(len (x2)):
    counter += x2[i] * y2[i]
```

```
2.6 s ± 20.7 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

```
counter = 0
for i in range(len (x2)):
    counter += (x2[i] * y2[i])
```

CHECK THE TWO ANSWERS AGREE

- The conclusion is that by using numpy's data structures and functions you can speed things up dramatically.
- numpy took about 9 milliseconds, whereas the for loop took about 100 times longer.

```
print ("With the numpy dot product the answer is", dot_p)
print ("With the for loop the answer is", counter)
```

```
With the numpy dot product the answer is -822.2796314998993
With the for loop the answer is -822.2796314999154
```

MATRICES

MATRICES

MATRIX TRANSPOSE

- In numpy the transpose is achieved through the ‘transpose’ function, which you can call in different ways, but the easiest is ‘.T’.

```
A = np.array([[3 , 2 ,  6], [ 4 , 7 , 5]])  #A vector of vectors.
print(A)
a_trans = A.T
print("\nThe transpose is:\n", a_trans)
```

```
[[3 2 6]
 [4 7 5]]

The transpose is:
 [[3 4]
 [2 7]
 [6 5]]
```

MATRIX MULTIPLICATION

MATRIX MULTIPLICATION

IMPLEMENTATION IN NUMPY

- Note the ‘@’ sign below to perform matrix multiplication. and not the usual ’*’.
- If the two matrices had the same dimensions, the ’*’ would do elementwise multiplication.

```
A = np.array([[3 , 2 ,  6], [ 4 , 7 , 5]])  #A vector of vectors.
B = np.array([[1 , 4 , 2 , 9], [4 , 2 , 1 , 3], [6 , 2 , 4 , 9]])

print(A @ B) # Note the "@" to multiply two matrices.
```

```
[[ 47  28  32  87]
 [ 62  40  35 102]]
```

MATRIX INVERSE

INVERTING A MATRIX WITH NUMPY

- We will invert the matrix $\begin{pmatrix} 1 & 2 & 1 \\ 1 & 1 & 0 \\ 1 & 0 & 0 \end{pmatrix}$.

```
C = np.array([[1 , 2 , 1], [1 , 1 , 0], [1 , 0 , 0]]) # Enter the matrix.
C_inv = np.linalg.inv(C) # The matrix inverse function.
print("Here's the inverse:\n",  C_inv)
# Confirm it really is the inverse:
print("\nHere's the identity matrix obtained from their product:\n", C @ C_inv)
```

```
Here's the inverse:
 [[ 0.  0.  1.]
 [-0.  1. -1.]
 [ 1. -2.  1.]]

Here's the identity matrix obtained from their product:
 [[1. 0. 0.]
 [0. 1. 0.]
 [0. 0. 1.]]
```

LEAST SQUARES

- Returning to regression, if we write the predictor variables in a matrix X and the outcome variable in a vector Y , then you can show that the Least Squares estimates of the regression slopes are given by:

$$\left(X^T X\right)^{-1} X^T Y.$$

REGRESSION CALCULATION

- We will use the 5K race data to illustrate.

```
import os
import numpy as np
np.set_printoptions(precision=5, suppress=True)
import pandas as pd
os.chdir('C:\\Users\\richardw\\Dropbox (Penn)\\Teaching\\477s2020\\DataSets')
race_data = pd.read_csv("Run5K.csv")
race_data['Finish_Time'] = race_data['Time_mins'] + race_data['Time_secs']/60 # Calculate the actual
  race time.
```

```
Y = np.array(race_data['Finish_Time'] ) # The outcome variable.
X = np.array(race_data['AGE']) # The AGE variable.
X = np.stack([np.ones(len(X)), X]).T  # We prepend a column of 1's to capture the constant term, and
  transpose the result.
print(X[0:5]) # Just look at the top 5 rows:
```

```
[[ 1. 21.]
 [ 1. 21.]
 [ 1. 20.]
 [ 1. 19.]
 [ 1. 19.]]
```

THE LEAST SQUARES CALCULATION

- Back in the class 7 notes we obtained 24.1633 for the intercept and 0.0688 for the AGE slope.
- Below we calculate these from the basic linear algebra operations and confirm that it works!

```
# This is (X^T X)^{-1} X^T Y

np.linalg.inv(X.T @ X) @ X.T @ Y
```

```
array([24.16335,  0.06884])
```

REPRESENTING A PICTURE AS A MATRIX

REPRESENTING A PICTURE AS A MATRIX

- If you consider a black and white picture as a grid of tiny black or white squares (pixels), you can represent it as a numeric matrix of zeros and ones.
- Below is a picture of my dog Sophy, and her reduction to a binary matrix.

SOPHY AS A MATRIX

```
from PIL import Image # PIL is an image processing library
os.chdir('C:\\Users\\richardw\\Dropbox (Penn)\\Teaching\\477s2020\\Notes\\images')
im = Image.open("sophy_bw.bmp") # Open up Sophy's image.
sophy = np.array(im) # Turn Sophy's image into a numpy array.
print(sophy.shape) # Confirm her shape.
sophy[128:160, 128:160] # Review a small piece of Sophy
```

```
(256, 256)
```

```
array([[False, False, False, ...,  True,  True,  True],
       [False, False, False, ...,  True,  True,  True],
       [False, False, False, ...,  True,  True,  True],
       ...,
       [ True,  True,  True, ...,  True,  True,  True],
       [ True,  True,  True, ...,  True,  True,  True],
       [ True,  True,  True, ...,  True,  True,  True]])
```

OPERATING ON SOPHY

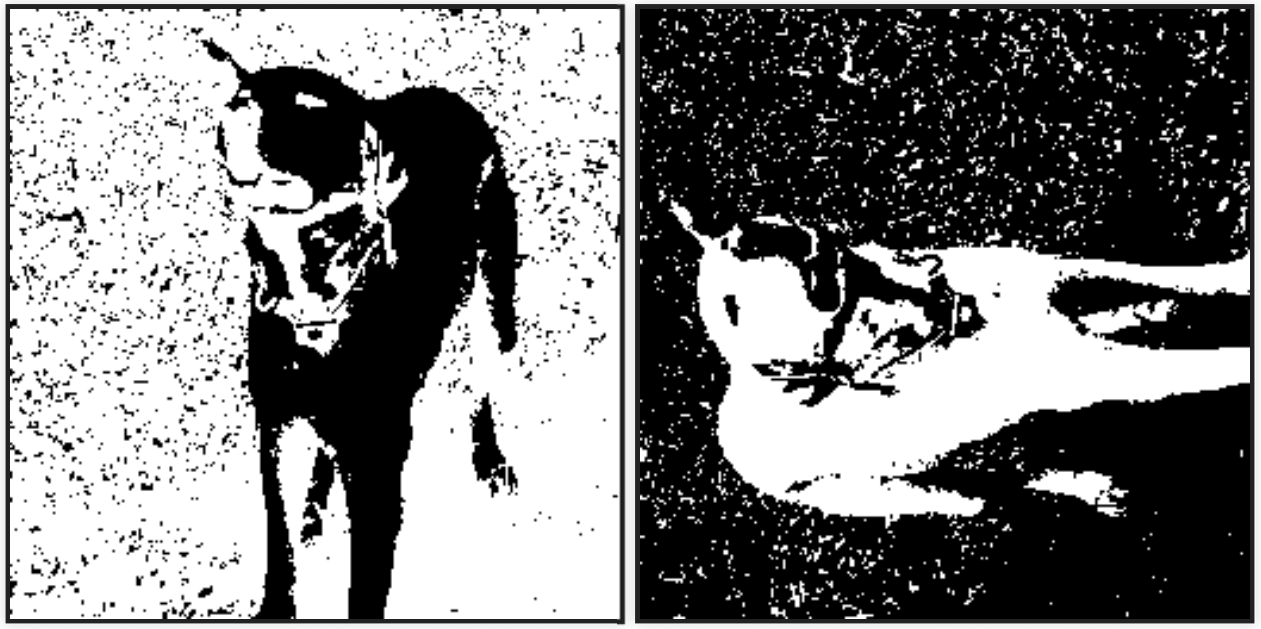
- If Sophy is a matrix, we can apply matrix operations to her.

```
# Negating Sophy
not_sophy = np.logical_not(sophy) # Turn True to False, and False to True.
img = Image.fromarray(not_sophy) # Make the image from the array.
#img.show()
#img.save('sophy_reverse.png')
```

```
# Transposing Sophy
t_sophy = sophy.T
img = Image.fromarray(t_sophy)
#img.show()
#img.save('sophy_transpose.png')
```

POOR SOPHY!

- Below we have Sophy's logical not (the negative of the image), and her transpose.



MONTE CARLO SIMULATION

MONTE CARLO SIMULATION

MONTE CARLO SIMULATION

- Monte Carlo simulations are probabilistic methods, that explore scenarios.
- The underlying drivers of the process, are drawn at random from pre-specified probability distributions.
- To create a realistic Monte Carlo simulation, you need to both understand how the process works, as well as knowing which probability distribution is relevant.
- Not everything is normally distributed!

SIMULATING THE STOCK MARKET

- The goal is to create a feasible range of values for what the S&P 500 will close at, at the end of the week.
- We will obtain historical returns on the market as inputs to the model.
- Based on the assumption that the future looks like the past, we randomly sample the historical returns, to simulate future returns.
- We do the sampling under the assumption, that daily returns are independent of one another.
- Then from the closing price today, we will simulate the closing price in 5 days, by drawing 5 daily returns at random from the historical series.

PROGRAM STRUCTURE

- Read in the price data.
- Calculate their returns.
- Simulate 5 returns.
- Calculate the resulting price 5 days into the future.

```
os.chdir('C:\\Users\\richardw\\Dropbox (Penn)\\Teaching\\477s2020\\DataSets')
sp_data = pd.read_csv("S_and_P.csv") # Read in the prices.
print(sp_data.head(5))
```

```
        Date   ClosePrice
0  1/2/2020  3257.850098
1  1/3/2020  3234.850098
2  1/6/2020  3246.280029
3  1/7/2020  3237.179932
4  1/8/2020  3253.050049
```

CALCULATING A RETURN

CALCULATING THE DAILY RETURN

```
returns = sp_data['ClosePrice'].pct_change()
print(returns[:5]) # The first element is missing because it has no prior price.
```

```
0         NaN
1   -0.007060
2    0.003533
3   -0.002803
4    0.004902
Name: ClosePrice, dtype: float64
```

GOING FROM A RETURN TO A PRICE

- Say yesterday's price was 3000 and today's return is 0.01, then what is the new price?
- The **increase** in price is 3000 * 0.01 = 30.
- So the new price is 3000 + 30 = 3030 = 3000 (1 + 0.01).
- In general if yesterday's price is P_{t-1} , then today's price is

$$P_t = P_{t-1} * (1 + R_t),$$

where R_t is today's return.

THE FIVE DAY RETURN

- Say we had returns of 0.01, 0.02, -0.01, -0.015 and 0.005 over a five day period, then their c effect on the price would be:

$$P_5 = P_0 * (1 + 0.01) * (1 + 0.02) * (1 - 0.01) * (1 - 0.015) * (1 + 0.005) = P_0$$

- We are now ready to simulate a week's price activity based on randomly sampling 5 histo returns.
- But first we need to drop the NaN in the returns' series.

```
returns.drop([0], inplace=True) # Drop the NaN element from the returns Series
returns[:5]
```

```
1   -0.007060
2    0.003533
3   -0.002803
4    0.004902
5    0.006655
Name: ClosePrice, dtype: float64
```

CALCULATE THE PRICE AT THE END OF THE WEEK

- First we randomly sample 5 returns and calculate the price multipliers:

```
np.random.seed(1234)
week_returns = np.random.choice(returns, size=5, replace=True) # A random draw of 5 returns.
print(week_returns) # Have a look at the sample.
price_mults = (1 + week_returns) # Calculate the price multipliers.
print(price_mults)
```

```
[-0.09511 -0.00824 -0.04336 -0.00565  0.01243]
[0.90489 0.99176 0.95664 0.99435 1.01243]
```

THE CUMULATIVE PRODUCT FUNCTION

- numpy has cumulative sum, and cumulative product functions to work with vectors.
- The cumulative sum of (3, 2, 5, 4) is (3, 5, 10, 14).
- The cumulative product of (3, 2, 5, 4) is (3, 6, 30, 120).

```
# The cumulative product of the price multipliers.
price_mults.cumprod() # Note the last element is 0.90489 * 0.99176 * 0.95664 * 0.99435 * 1.01243 =
  0.86428
```

```
array([0.90489, 0.89743, 0.85852, 0.85367, 0.86428])
```

```
# If the initial price were 3000 then after 5 days the price would be
3000 * price_mults.cumprod()[-1] # This is 3000 * 0.86428 = 2592.84.
```

```
2592.8395299664826
```

CREATE THE MONTE CARLO SIMULATION

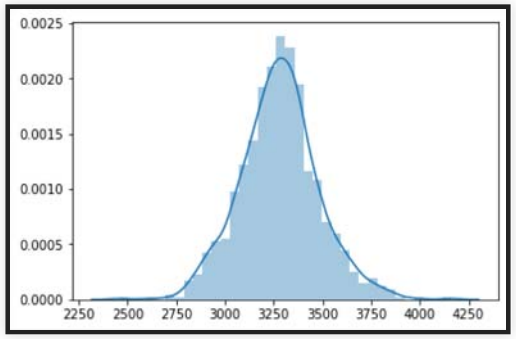
- Essentially we just repeat the previous calculations, but now inside a 'for loop', keeping track of the outcome of each iteration.
- The results will be in the vector 'price_vec'.

```
np.random.seed(1234)
init_price = 3271.12
sim_size = 1000
price_vec = np.empty(sim_size)

for i in range(sim_size):
    week_returns = np.random.choice(returns, size=5, replace=True)
    price_mults = (1 + week_returns)
    final_price = init_price * price_mults.cumprod()[-1]
    price_vec[i] =  final_price

price_vec[:10] # Have a look at just the first ten prices.
```

```
array([2827.16308, 3308.59344, 3489.5888 , 3161.87936, 3469.18856,
       3204.44453, 3285.68698, 3201.19699, 3743.90089, 2916.76575])
```

REVIEW THE DISTRIBUTION OF PRICES

```
import seaborn as sns
sns.distplot(price_vec);
```



CALCULATE PERCENTILES OF THE PRICE DISTRIBUTION

- Taking advantage of what we know from pandas, below are the 2.5th and 97.5th percentiles of the price distribution:

```
pd.Series(price_vec).quantile([0.025,0.975])
```

```
0.025    2884.708284
0.975    3694.499557
dtype: float64
```

- This gives a 95% prediction interval for the price at the end of the week, as being between 2879 and 3682.

WORKING WITH THE WHOLE 5 DAY PRICE HISTORY

- Rather than just looking at the price on day 5, we could store the whole price history, P_1, P_2, P_3, P_4, P_5 .
- To acheive this we will use a matrix with 5 columns, where the result of each iteration is stored in a row.

```
np.random.seed(12345)
init_price = 3271.12
sim_size = 1000
price_mat = np.empty((sim_size, 5)) # Note the matrix container.

for i in range(sim_size):
    week_returns = np.random.choice(returns, size=5, replace=True)
    week_mults = (1 + week_returns)
    prices = init_price * week_mults.cumprod()
    price_mat[i,:] = prices # Store the 5 prices in a single row.

price_mat[0,:] # Have look at the first row.
```

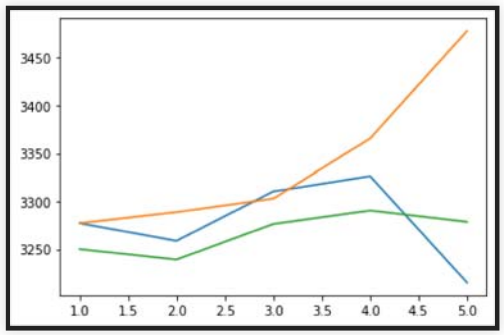
```
array([3277.15042, 3258.65547, 3310.40855, 3326.33934, 3214.86165])
```

VISUALIZE THE PRICE PATH

- Below we plot the 5 prices, for the first three runs of the simulation using seaborn's 'lineplot' function.

```
sns.lineplot(x= range(1,6), y = price_mat[0,:])
sns.lineplot(x= range(1,6), y = price_mat[1,:])
sns.lineplot(x= range(1,6), y = price_mat[2,:])
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x49038548>
```



VIEWING ALL THE PRICE HISTORIES

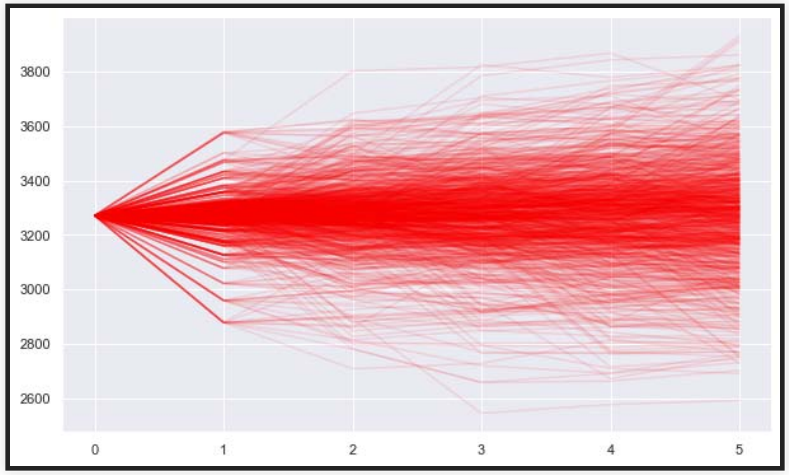
- Below we use a for loop to iterate over the rows of the entire results matrix.
- By using transparency in the graph (alpha = 0.1) we can get a sense of the density of the prices.
- But first we will add a column at the beginning of the price matrix, to reflect the starting price:

```
price_mat_new = np.insert(price_mat, 0, init_price, axis=1)
price_mat_new
```

```
array([[3271.12   , 3277.15042, 3258.65547, 3310.40855, 3326.33934,
        3214.86165],
       [3271.12   , 3277.15042, 3288.72981, 3302.89575, 3365.52658,
        3478.3256 ],
       [3271.12   , 3249.94476, 3238.99666, 3276.32552, 3290.43803,
        3278.59147],
       ...,
       [3271.12   , 3078.30691, 3118.5122 , 3154.4525 , 2776.42117,
        2797.71772],
       [3271.12   , 3258.75994, 3304.11555, 3310.2068 , 3329.23083,
        3374.67143],
       [3271.12   , 3409.16974, 3486.99886, 3427.5321 , 3485.35984,
        3496.951  ]])
```

PLOTTING ALL THE PRICE HISTORIES

```
sns.set(rc={'figure.figsize':(10, 6)})
for i in range(sim_size):
  sns.lineplot(x= range(0,6), y = price_mat_new[i,:],color="red",alpha = 0.10)
```



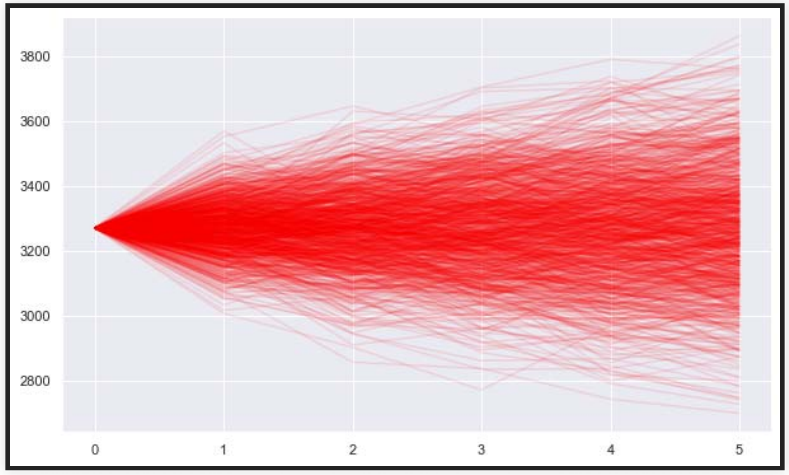
SAMPLING FROM A MODEL

- We have been sampling from the historical returns directly.
- An alternative would be to fit a model to the historical returns, and sample from the model.
- Below, we sample from a normal model, fit to the historical returns, which results in a smoother distribution.
- We use the mean and standard deviations of the historical returns for the parameters of the normal distribution.

```
np.random.seed(12345)
init_price = 3271.12
sim_size = 1000
price_mat = np.empty((sim_size, 5))

for i in range(sim_size):
    week_returns = np.random.normal(returns.mean(), returns.std(), size=5)
    week_mults = (1 + week_returns)
    prices = init_price * week_mults.cumprod()
    price_mat[i,:] = prices
```

SAMPLING FROM THE NORMAL MODEL

```
price_mat_new = np.insert(price_mat, 0, init_price, axis=1)

sns.set(rc={'figure.figsize':(10, 6)})
for i in range(sim_size):
  sns.lineplot(x= range(0,6), y = price_mat_new[i,:],color="red",alpha = 0.105)
```



CARD COUNTING

CARD COUNTING

CARD COUNTING

- In some card games of chance, it is possible for a player to gain an advantage if they can remember all the cards dealt so far.
- It isn't necessary to remember the suit and value of every card dealt, but rather assign a "count" to it.
- When the player keeps a cumulative sum of the 'count' they can track the state of the card deck, so if for example the count is low, which means few picture cards have been dealt, the probability of the next card being a picture card, is higher than when the cumulative count is high.
- The player can exploit this change in probability to their advantage as they gamble.
- Essentially they are exploiting the fact that successive draws of a card from a deck, are not statistically independent.
- For example, the probability that the second card dealt is the ace of hearts, given that the first card dealt was the ace of hearts, is 0 not 1/52!

A CARD COUNTING SYSTEM

- One card counting system assigns the following points to each card:

cardvalues	**ACE**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**J**	**Q**	**K**
cardcounts	-1	0.5	1	1	1.5	1	0.5	0	-0.5	-1	-1	-1	-1

- Example: say the first 5 cards dealt were the 5D, 9H, QC, 4D, 10S, then their counts would be 1.5, -0.5, -1, 1, -1.
- The cumulative count is 1.5, 1, 0, 1, 0.

AN ADVANTAGE SIGNAL

- We need to define a cut-off for the cumulative card count, so that if it rises above a certain level, a “signal” is generated, and the player should then change their strategy.
- The issue is that if we make the cut-off too high, then the advantage will be large, but it will very rarely be triggered.
- On the other hand, if we make it too low, it will be triggered frequently, but the advantage will be small.
- So a first step in any strategy is, for a given advantage cut-off to find the probability that it gets triggered.
- Our specific question: given the card counting system on the previous slide, what is the probability that the cumulative count rises strictly about 5, within the first 30 cards dealt (including the 30th)?
- We will set up a Monte Carlo simulation to empirically estimate this probability.

APPROACH

- Shuffle a deck of cards.
- Replace the card values with their respective counts.
- Create a cumulative sum of the counts.
- Look to see if the cumulative count rises above 5 within the first 30 cards.
- First code one iteration for understanding, then put the code in a loop.

CHECKING THE CARD COUNT CONDITION

```
# Create the deck (we can simply use the counts vector to simplify the program)
card_counts = np.array([-1,  0.5,  1, 1, 1.5, 1, 0.5, 0, -0.5, -1, -1, -1, -1])
card_counts = np.repeat(card_counts, 4)
print(card_counts)
print("The length of the vector is", card_counts.shape)
np.random.shuffle(card_counts)
print(card_counts)
card_counts.cumsum()
if max(card_counts.cumsum()[:30] > 5):
    print("Signal generated")
else:
    print("No signal")
```

```
[-1.  -1.  -1.  -1.   0.5  0.5  0.5  0.5  1.   1.   1.   1.   1.   1.
  1.   1.   1.5  1.5  1.5  1.5  1.   1.   1.   1.   0.5  0.5  0.5  0.5
  0.   0.   0.   0.  -0.5 -0.5 -0.5 -0.5 -1.  -1.  -1.  -1.  -1.  -1.
 -1.  -1.  -1.  -1.  -1.  -1.  -1.  -1.  -1.  -1. ]
The length of the vector is (52,)
[ 0.5 -1.  -1.  -1.   1.5  1.   1.   1.   1.  -1.   1.  -1.  -0.5 -1.
  1.   1.  -1.  -1.   0.  -1.   0.   0.5 -1.   0.  -1.  -1.  -0.5  1.
 -1.   1.   0.5 -1.   1.5  0.5 -1.  -1.   0.5 -1.   1.5 -1.  -0.5  1.
  1.  -1.   0.5  0.   0.5 -1.   1.  -0.5  1.5  0.5]
No signal
```

THE MONTE CARLO LOOP

```
np.random.seed() # Set the random number seed.
sim_size = 1000 # Decide on the size of the simulation.
results_vec = np.empty(sim_size) # Create an empty container for the results.
card_counts = np.array([-1,  0.5,  1, 1, 1.5, 1, 0.5, 0, -0.5, -1, -1, -1, -1])
card_counts = np.repeat(card_counts,4) # Make the card counts vector.

for i in range(sim_size): # Start the Monte Carlo loop.
    np.random.shuffle(card_counts) # Shuffle the card deck.
    if max(card_counts.cumsum()[:30]) > 5: # Look to see if the max of the cumsum is greater than 5.
        results_vec[i] = 1
    else:
        results_vec[i] = 0
print("The estimated  probability of an advantage signal is", sum(results_vec)/sim_size) # Calculate
  the proportion.
```

```
The estimated  probability of an advantage signal is 0.18
```

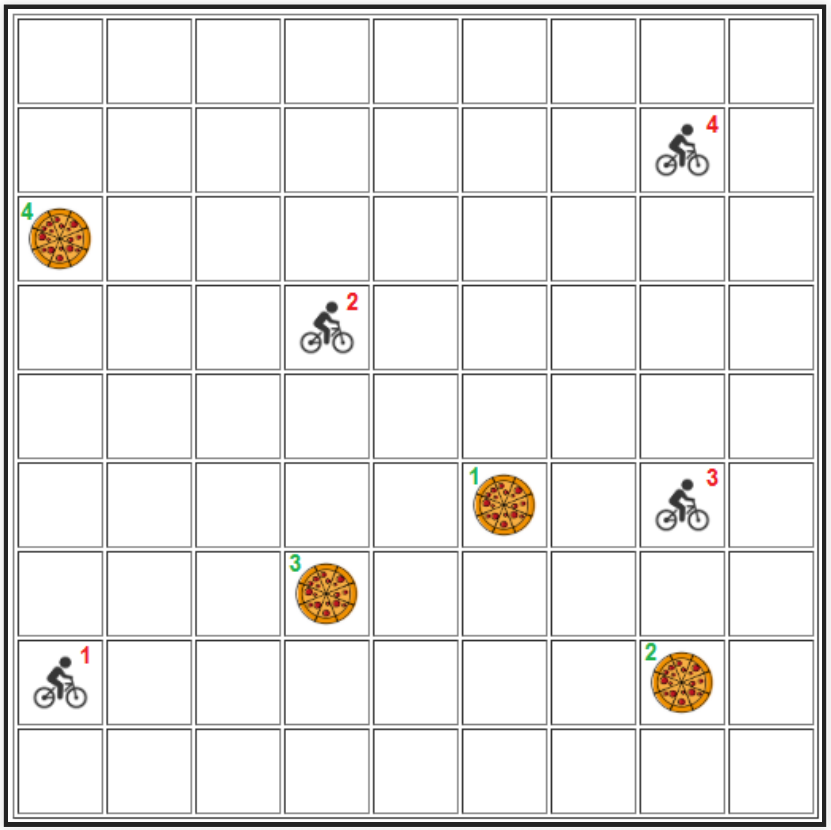
OPTIMIZATION

OPTIMIZATION

OPTIMALLY ROUTING BIKES TO ORDERS

- We have 4 bicyclists and 4 orders to be picked up.
- How should we assign bikes to orders to minimize the average waiting time?
- The example below has 4 cyclists randomly located in a 9 by 9 grid with 4 orders.
- The bottom left hand corner of the grid is (1,1) and the top right, (9,9).
- The cyclists are at [1,2], [4,6], [8,4] and [8,8]
- The orders are at [6,4], [8,2], [4,3], [1,7].

THE CITY GRID



THE DISTANCE MATRIX

- The key component of the solution is the distance matrix, which finds the distance between each cyclist and each order.
- We will use the “city block” distance which counts the number of streets from one location to another.
- For example, the distance between cyclist 1, [1,2], and order 1, [6,4], is 5 + 2 = 7.
- Below is the complete distance matrix:

```
[[ 7.  7.  4.  5.]
 [ 4.  8.  3.  4.]
 [ 2.  2.  5. 10.]
 [ 6.  6.  9.  8.]]
```

THE ASSIGNMENT OF CYCLISTS TO ORDERS

- An assignment of cyclists to orders, assigns each and every cyclist to exactly one order.
- For example, if cyclist 1 went to order 2, cyclist 2 to order 4, cyclist 3 to order 1 and cyclist 4 to order 3, we could write this as the matrix:

```
[[ 0  1  0  0]
 [ 0  0  0  1]
 [ 1  0  0  0]
 [ 0  0  1  0]]
```

  - Note that every row has exactly one 1, and every column has exactly one 1.
  - The cost of the above assignment would be 7 + 4 + 2 + 9 = 22.
  - The goal is to find the assignment that minimizes the cost.

OPTIMIZATION LIBRARY

- We will use the scipy.optimize library which includes the solution to this problem.
- The relevant function is called ‘linear_sum_assignment’ and it takes as input the distance matrix.
- It returns the optimal assignment of rows to columns.
- In the above example the optimal assignment is:

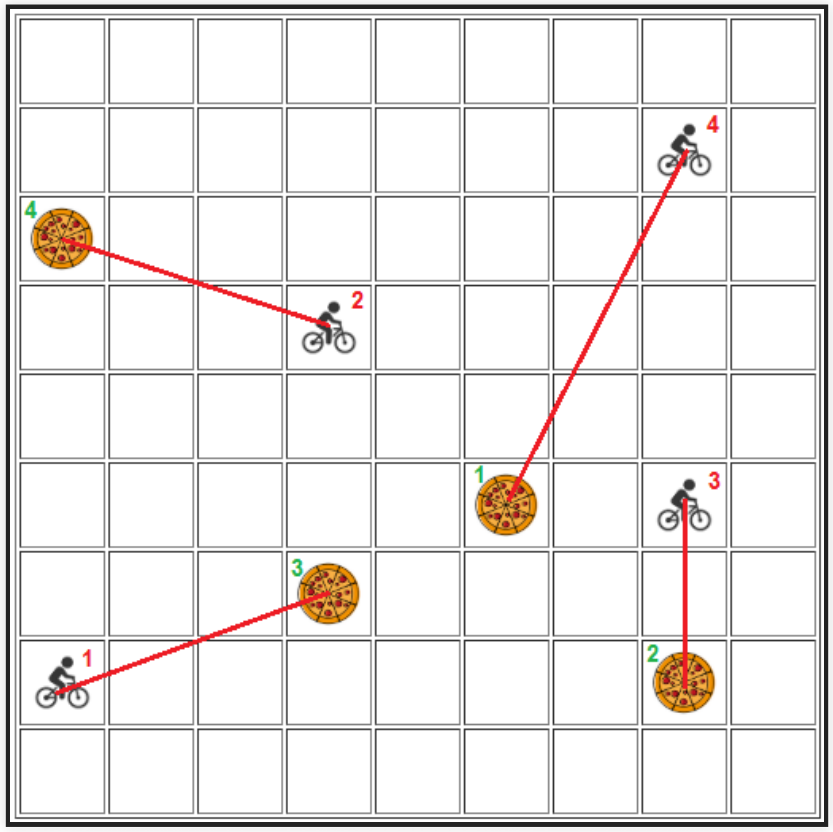
```
rows    = [1 2 3 4]
columns = [3 4 2 1]
```

- This means row 1 goes to column 3, row 2 to column 4, rows 3 to column 2 and row 4 to column 1.

```
[[ 0  0  1  0]
 [ 0  0  0  1]
 [ 0  1  0  0]
 [ 1  0  0  0]]
```

- The cost of the assignment is 4 + 4 + 2 + 6 = 16.

THE OPTIMAL ASSIGNMENT SOLUTION



SET UP THE INPUTS

```
from scipy.optimize import linear_sum_assignment
# Make  function that calculates grid distance between points.
def distance (x,y):
    return sum(np.abs(x - y))
# Create arrays to hold bike and order locations.
bicycle_location    = np.array([[1,2], [4,6], [8,4], [8,8]])
restaurant_location = np.array([[6,4], [8,2], [4,3], [1,7]])

dist_mat = np.zeros([4,4]) # Start with an empty matrix.
dims = dist_mat.shape
print (dist_mat)
```

```
[[0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]]
```

CALCULATE THE DISTANCE MATRIX

- Nested for loops are being used to calculate the elements of the distance matrix.

```
for i in range(dims[0]):
    for j in range(dims[1]):
        dist_mat[i,j] = distance(bicycle_location[i,:], restaurant_location[j,:] )
print(dist_mat)
```

```
[[ 7.  7.  4.  5.]
 [ 4.  8.  3.  4.]
 [ 2.  2.  5. 10.]
 [ 6.  6.  9.  8.]]
```

RUN THE ASSIGNMENT ALGORITHM

```
row_ind, col_ind = linear_sum_assignment(dist_mat) # The critical line, where the optimization happens.
print ("Optimal assignment is:")
print(row_ind) # Don't forget Python starts counting from 0,
print(col_ind) # but in the grid, I started counting from 1.
print ("Minimum cost is:")
print(dist_mat[row_ind, col_ind].sum()) # This adds up the elements in the distance matrix
                                        # corresponding to the optimal assignment.
```

```
Optimal assignment is:
[0 1 2 3]
[2 3 1 0]
Minimum cost is:
16.0
```

SUMMARY

SUMMARY

- An overview of scientific computing and numpy:
- Vectors, matrices and arrays.
- Images as numbers.
- Monte Carlo simulation.
- Optimization algorithms.