Final Project Discussion (2/8)

Learning Objectives:

- Describe what "good" means for each stakeholder in the Marshall course scheduling case. (Model)
- Identify appropriate metrics/analyses to measure each notion of "goodness." (Model)
- Disecting, explaining and modifying the Python code in Lab 2 solutions. (Code)

Project material posted on Blackboard:

- Overview of project and deliverables (more detailed description of each deliverable and grading rubric will be posted with the submission link.) (Assignments tab)
- Currently available data sets and description of data. (Assignments tab)
- Visuals used by Shannon and Hal in their in-class discussion. (Content tab)
- Team assignment spreadsheet. (Assignments tab)
- Free supplementary resources on basic Python. (Content tab)

Defining and Measuring "Goodness"

In-class exercise:

Identify what a good outcome of the scheduling system looks like to each of the following stakeholders: students, faculty, program, department, management (i.e. Marshall leadership, USC registrar, finance office).

For each outcome, find an appropriate metric or analysis to quantify or visualize it from data. The exact form of this is open ended: for example, it can be in the form of a single number for each stakeholder, or in the form of a graph or visual that illustrates the outcome. Essentially, any output of a data analysis counts. When conducting prescriptive analytics, a good metric/analysis for an outcome is CASE:

- **Computable** from available data and reasonable supplemental assumptions (in cases when certain data could be collected but is currently unavailable, you may fill in the gap for now by reasonable assumption.)
- **Actionable**: within our power to affect. For example, the number of classes scheduled in late evenings is within our power to affect, but not the number of PM MBA students or the number of students who have full time jobs.
- **Simple** to understand and interpret. Since there are many stakeholders, it is important that the analysis is transparent and understandable.
- **Enlightening** for the underlying outcome one is measuring. There should be a close relationship with the underlying outcome to be measured. For example, if the outcome is high utilization of 11-5pm slots, then a good metric in this category would be the percentage of times that is used up on average in JKP, but a bad metric would be the average number of classes that take place in JKP during this time, since classes have variable lengths and one can have many short classes but still have large gaps of free time in between.

Learning from Code Examples

When stumbling across unfamiliar code examples, the following steps can be taken to digest the example and learn from it. It has the acronym DEMO:

- **D**isect the code and break it up into small chunks, and "look inside" the working of the code by printing intermediary computations and looking up Python help or online documentation.

- **E**xplain in your own words what each small chunk of code does and what the overall piece of code does.
- **MO**dify each small chunk of code and apply it in new contexts, so as to make this coding construction your own. (It's like learning new vocabulary from example sentences.)

Digesting Lab 2 Solutions

We apply these steps to learn from the Lab 2 solutions: http://nbviewer.jupyter.org/github/pengshi46/usc-dso-570/blob/master/Labs/Lab2/Lab%202-Solution.ipynb?flush_cache=true You can click this to open in a new browser and refer to it when needed.

Let us begin with the first few lines.

```
from scipy.stats import norm,uniform
import numpy as np
import matplotlib.pyplot as plt
distN1=norm(200,60)
distN2=norm(60,20)
distBuyTime1=uniform(loc=0,scale=180)
distBuyTime2=uniform(loc=0,scale=14)
distValue1=norm(200,80)
distValue2=norm(350,120)
inventory=100
fixedCost=20000
```

Disecting

- Look through the code and look up unfamiliar commands or Syntax. You may want to Google the `scipy.stats` package, the `numpy` package, or the `matplotlib.pyplot` package to gain a general description. Moreover, you may copy the code into a new cell, and type `help(norm)` or `norm?` to obtain information on the `norm` module imported from the `scipy.stats` package, and similarly for the `uniform` module.
- According to `help(norm)`, the first parameter (called `loc`) specifies the mean and the second one (called `scale`) specifies the standard deviation.
- According to `help(uniform)`, the first parameter (`loc`) specifies where it begins and the second one (`scale`) the length. So `uniform(3,2)` is a uniform distribution from 3 to 3+2=5.

Explaining

- Explain what each line does. You may want to copy the code and write comments as below.

```
[10]: # Import statements
      from scipy.stats import norm,uniform     # Import norm and uniform from scipy.stats
      import numpy as np                       # Import the numpy package and call it np
      import matplotlib.pyplot as plt          # Import the matplotlib.pyplot package and call

      # Defining distributions
      distN1=norm(200,60)            # Define a normal distribution with mean 200 and stdev 60
      distN2=norm(60,20)             # Define a normal dist. with mean 60 and stdev 20.
      distBuyTime1=uniform(loc=0,scale=180) # Define a uniform distribution from 0 to 180
```

```
distBuyTime2=uniform(loc=0,scale=14)  # Define a uniform distribution from 0 to 14
distValue1=norm(200,80)       # Define a normal dist. with mean 200 and stdev 80
distValue2=norm(350,120)      # Define a normal dist. with mean 350 and stdev 120

# Setting parameters
inventory=100     # number of tickets
fixedCost=20000   # cost of flight
```

Modifying

Exercise:

- Import the `norm` package and define a normal distribution, called `X`, with mean -100 and standard deviation 50.
- Import the `uniform` package and define a uniform distribution, called `Y`, between -10 and 30.

Answers:

```
[19]: from scipy.stats import norm
      X=norm(-100,50)
      X=norm(loc=-100,scale=50) # Equivalent

      from scipy.stats import uniform
      Y=uniform(-10,40)
      Y=uniform(loc=-10,scale=40) # Equivalent
```

Generating a Scenario

We next digest the `generateScenario()` function:

```
def generateScenario():
    '''Generate a two row numpy array, with the first row being the sorted buy times,
    from earliest to latest. The second row is the corresponding maximum willingness to
    pay from each customer'''
    num1=int(max(0,distN1.rvs()))
    num2=int(max(0,distN2.rvs()))
    buyTimes1=distBuyTime1.rvs(size=num1)
    buyTimes2=distBuyTime2.rvs(size=num2)
    values1=distValue1.rvs(size=num1)
    values2=distValue2.rvs(size=num2)
    data=(np.vstack([np.hstack([buyTimes1,buyTimes2]),np.hstack([values1,values2])]))
    return data[:,(-data[0,:]).argsort()]
```

This function can be broken down into 3 smaller parts, which we disect, explain, and modify below. For brevity we will not differentiate each step as clearly as before but will do them together.

Generate random numbers

```
num1=int(max(0,distN1.rvs()))
num2=int(max(0,distN2.rvs()))
buyTimes1=distBuyTime1.rvs(size=num1)
```

```
buyTimes2=distBuyTime2.rvs(size=num2)
values1=distValue1.rvs(size=num1)
values2=distValue2.rvs(size=num2)
```

As explained in earlier classes, the `rvs` function draws random samples from the given distribution. So `distN1.rvs()` draws one random number from the `distN1=norm(200,60)` distribution (normal distribution with mean 200 and stdev 60). When specifying the `size` parameter, this generates an array of independent random samples with the given size. The `max(0, )` command makes sure that the number of customers of segment 1 is not negative, and the `int( )` command rounds down the fractional number into an integer.

Exercise:

- generate a normal random variables with mean 300 and standard dev 100 and round it down to nearest integer and make it zero if negative.
- generate an array of 5 uniformly random numbers between 10 and 14.

Answer: (In the following, we combine the distribution declaration and the rvs() in one line to illustrate how they can be combined.)

```
[25]: from scipy.stats import norm,uniform
      a=int(max(0,norm(300,100).rvs()))
      b=uniform(10,4).rvs(size=5)
      print(a,b)
```

```
380 [ 10.27716125  13.54311176  13.6163318   13.51581892  12.57043452]
```

Stacking arrays

The next line in the `generateScenario()` function is:

```
data=(np.vstack([np.hstack([buyTimes1,buyTimes2]),np.hstack([values1,values2])]))
```

You can look up the `np.vstack` and `np.hstack` functions using Python help. (Type `help(np.vstack)` or `np.vstack?` in a code cell and execute.) The above line stacks the arrays so that buyTimes1 and buyTimes2 are horizontally stacked, values1 and values2 are horizontally stacked, and all the buyTimes and the values are vertically stacked. The end result is a two row array of buyTimes and values.

The following code segment was used in class to illustrate creating `numpy` arrays and stacking them.

```
[22]: import numpy as np
      a=np.array([3,2,1])
      b=np.arange(3)
      c=np.zeros(2)
      print('a=',a)
      print('b=',b)
      print('c=',c)

      print('np.hstack((a,b))=\n',np.hstack((a,b)))
      print('np.vstack((a,b))=\n',np.vstack((a,b)))
```

```
a= [3 2 1]
b= [0 1 2]
c= [ 0.  0.]
np.hstack((a,b))=
 [3 2 1 0 1 2]
np.vstack((a,b))=
 [[3 2 1]
 [0 1 2]]
```

Exercise:

- create the four one-dimensional arrays `[1,2,3]` `[4,5,6,7]` `[3,2,1]` `[7,6,5,4]` and stack them into a three row array so that the first row is `1,2,3,4,5,6,7` and second row is `7,6,5,4,3,2,1`, and the third row is `1,2,3,7,6,5,4`.
- Horizontally stack the above array with itself.

Answer: (the `\n` symbol in the print statement is for new line)

```
[28]: a=np.array([1,2,3])
      b=np.array([4,5,6,7])
      c=np.array([3,2,1])
      d=np.array([7,6,5,4])
      firstRow=np.hstack((a,b))
      secondRow=np.hstack((d,c))
      thirdRow=np.hstack((a,d))
      answer=np.vstack((firstRow,secondRow,thirdRow))
      print('stacked 1=\n',answer)
      print('stacked 2=\n',np.hstack((answer,answer)))
```

```
stacked 1=
 [[1 2 3 4 5 6 7]
 [7 6 5 4 3 2 1]
 [1 2 3 7 6 5 4]]
stacked 2=
 [[1 2 3 4 5 6 7 1 2 3 4 5 6 7]
 [7 6 5 4 3 2 1 7 6 5 4 3 2 1]
 [1 2 3 7 6 5 4 1 2 3 7 6 5 4]]
```

Sorting a 2-D array by a given row/column

The `generateScenario()` function returns the following object

```
data[:,(-data[0,:]).argsort()]
```

This is a complex expression and can be used to illustrate the **disecting step** in understanding code. First, we know that the `data` variable contains a two rowed 2-dimensional python array, so instead of using the actual generated data, which is large, let us use a smaller array for the disecting step. See the original data and the output of the above line below.

```
[63]: data=np.array([[3,1,8,9,5],[6,7,2,4,10]])
      print('data=\n',data)
      print('data[:,(-data[0,:]).argsort()]=\n',data[:,(-data[0,:]).argsort()])
```

```
data=
 [[ 3  1  8  9  5]
 [ 6  7  2  4 10]]
data[:,(-data[0,:]).argsort()]=
 [[ 9  8  5  3  1]
 [ 4  2 10  6  7]]
```

As can be seen, the complex command sorts the original two rowed array by the first row, from largest to smallest. This was what we wanted in the simulation so as to sort the customers from those who come 180 days before to those who come just before arrival, while keeping the value for each customer. (If we had sorted the whole array, we would have sorted both rows and not kept the correspondence between each valuation and each time of arrival.)

The following block of code breaks down the line into smaller components and print what each component does. An explanation of each piece is given in the explanatory print statements. (The `\t` in print statments are for tab.) **Read the output first, as that is most clear.**

```
[64]: print('data=\n',data)
      print('\nGet first row of data')
      print('\tdata[0,:]=',data[0,:])

      print('\nGet sorted order of first row: i.e. index of smallest element, index of second
      sortedOrder=data[0,:].argsort()
      print('\tsortedOrder=data[0,:].argsort()=',sortedOrder)
      print('\nReversing the order by sorting the negative of first row')
      reverseOrder=(-data[0,:]).argsort()
      print('\treverseOrder=(-data[0,:]).argsort()=',reverseOrder)
      print('\nGet all rows of original data, but reshuffle the columns according to sorted o
      print('\ndata[:,reverseOrder]=\n',data[:,reverseOrder])
```

```
data=
 [[ 3  1  8  9  5]
 [ 6  7  2  4 10]]

Get first row of data
        data[0,:]= [3 1 8 9 5]

Get sorted order of first row: i.e. index of smallest element, index of second smallest, ...
        sortedOrder=data[0,:].argsort()= [1 0 4 2 3]

Reversing the order by sorting the negative of first row
        reverseOrder=(-data[0,:]).argsort()= [3 2 4 0 1]

Get all rows of original data, but reshuffle the columns according to sorted order

data[:,reverseOrder]=
 [[ 9  8  5  3  1]
 [ 4  2 10  6  7]]
```

The alternative command `np.sort`, would sort every row by itself. To indicate which axis to sort by, supply the axis argument. `axis=0` meaning sorting by row, and `axis=` meaning sorting

by column (the default would be to sort by the last axis. For 2-D arrays, that would be sorting by the columns). See the output below.

```
[70]: print('original data: \n',data)
      print('\nSorting each row by columns: np.sort(data):\n',np.sort(data))
      print('\nSorting each row by columns (equivalent): np.sort(data,axis=1):\n ', np.sort(d
      print('\nSorting each column by the row: np.sort(data,axis=0):\n ', np.sort(data,axis=0
```

```
original data:
 [[ 3  1  8  9  5]
 [ 6  7  2  4 10]]

Sorting each row by columns: np.sort(data):
 [[ 1  3  5  8  9]
 [ 2  4  6  7 10]]

Sorting each row by columns (equivalent): np.sort(data,axis=1):
  [[ 1  3  5  8  9]
 [ 2  4  6  7 10]]

Sorting each column by the row: np.sort(data,axis=0):
  [[ 3  1  2  4  5]
 [ 6  7  8  9 10]]
```

Exercise:

- Define the array `[[1,3,4],[2,1,3],[3,2,1]]` and sort it in increasing order by row 0.
- Sort it in decreasing order by row 1.
- Sort every row in by itself in increasing order using `np.sort()`.
- Sort every column by itself in increasing order using `np.sort()`.
- Sort it in reverse order by column 1.
- Sort the array `[2,1,3,10,9,8]` according to increasing order of the array `[10,8,6,4,3,1]`.

Answer:

```
[80]: a=np.array([[1,3,4],[2,1,3],[3,2,1]])
      print('Original array a=\n',a)
      print('Sorted in increasing order by row 0:\n',a[:,a[0].argsort()])
      print('Sorted in decreasing order by row 1:\n',a[:,(-a[1]).argsort()])
      print('Every row sorted by itself:\n',np.sort(a))
      print('Every column sorted by itself:\n',np.sort(a,axis=0))
      print('Sorted in reverse order by column 1\n',a[(-a[:,1]).argsort(),:])
      b=np.array([2,1,3,10,9,8])
      c=np.array([10,8,6,4,3,1])
      print('Array b={0}, c={1}, b[c.argsort()]={2}'.format(b,c,b[c.argsort()]))
      print('This reverses array b because c is in reverse order.')
```

```
Original array a=
 [[1 3 4]
 [2 1 3]
 [3 2 1]]
```

```
Sorted in increasing order by row 0:
 [[1 3 4]
 [2 1 3]
 [3 2 1]]
Sorted in decreasing order by row 1:
 [[4 1 3]
 [3 2 1]
 [1 3 2]]
Every row sorted by itself:
 [[1 3 4]
 [1 2 3]
 [1 2 3]]
Every column sorted by itself:
 [[1 1 1]
 [2 2 3]
 [3 3 4]]
Sorted in reverse order by column 1
 [[1 3 4]
 [3 2 1]
 [2 1 3]]
Array b=[ 2  1  3 10  9  8], c=[10  8  6  4  3  1], b[c.argsort()]=[ 8  9 10  3  1  2]
This reverses array b because c is in reverse order.
```

Simulating Scenarios

A scenario as generated above is this 2 rowed table, with the first row being sorted buy times of each customer (in days before departure) and second row being corresponding valuations of each. There are three functions in Lab 2 that are used to simulate the profit in a given scenario, and each function corresponds to one of the three pricing policies. Besides taking the scenario as an input (called `data`), each function also takes the parameters of the pricing policy.

- For constant price, the `price` parameters is the price. (see the `simulatePrice` function)
- For RM1, the `price1` parameter is the initial price offered, and `price2` is the price in the last 14 days. (see the `simulateRM1` function)
- For RM2, `price1` is the initial price, `price2` is the new price after `quantity` number of tickets are sold. (see the `simulateRM2` function)

The overall logic is that the functions calculate the profit for each scenario under a given paramter. The later code searches through a large set of parameters and find one with the best average profit under 100 scenarios.*

Let us apply the disect, explain, and modify method to digest each function in sequence.

Constant Price

```
def simulatePrice(data,price):
    '''Given the a simulated scenario from generateScenario() (stored in the data variable),
    and given a price, return the profit from implementing constant price policy with this pr
    return min(inventory,np.sum(data[1]>=price))*price-fixedCost
```

As you will learn by using python help on `inventory` and `fixedCost`, these are parameters set in the beginning of the lab, and equal to 100 and 20000 respectively. The `min` function

takes the minimum of the given argument. The most difficult to understand expression here is `np.sum(data[1]>=price)`, which we disect below.

```
[46]: data1=np.array([3,5,2,1,4,3])
      print('data1 is',data1)
      print('data1>=3 is',data1>=3)
      print('np.sum(data1>=3) is',np.sum(data1>=3))
```

```
data1 is [3 5 2 1 4 3]
data1>=3 is [ True  True False False  True  True]
np.sum(data1>=3) is 4
```

As can be seen, when we take a numpy array and apply >= with a number, we get a True/False array of whether each element of the original array is greater than or equal to that given number. Finally, `np.sum` sums the True/False array, with True interpreted as 1 and False as 0. The end result is simply to count how many elements are greater than or equal to 3.

In context of `simulatePrice`, the line `np.sum(data[1]>=price)` counts how many people have valuations greater than or equal to the price, take the minimum of this and the number of available tickets to obtain the number of tickets sold, multiply by the price and minus the fixed cost to obtain the profit.

Exercise: Declare an array `a=np.array([[1,3,4,5,6],[3,2,7,1,2]])` and use `np.sum` to count

- how many entries are less than 3.
- how many entries are less than or equal to 3.
- how many entries in row 0 are less than or equal to 3.
- how many entries in column 1 are greater than 2.
- the total sum of the entries.
- the total sum of entries in column 0

Answer:

```
[51]: a=np.array([[1,3,4,5,6],[3,2,7,1,2]])
      print('a=\n',a)
      print('# entries < 3:',np.sum(a<3))
      print('# entries <= 3:',np.sum(a<=3))
      print('# entries in row 0 <=3:', np.sum(a[0]<=3))
      print('# entries in column 1 >2:',np.sum(a[:,1]>2))
      print('Sum of entries:',np.sum(a))
      print('Sum of entries in column 0:',np.sum(a[:,0]))
```

```
a=
 [[1 3 4 5 6]
 [3 2 7 1 2]]
# entries < 3: 4
# entries <= 3: 6
# entries in row 0 <=3: 2
# entries in column 1 >2: 1
Sum of entries: 34
Sum of entries in column 0: 4
```

RM 1 Policy: Increase Price in Last 14 Days

```
def simulateRM1(data,price1,price2):
    '''Given the simulated scenario from generateScenario() (stored in the data variable),
    as well as the price before 14 days to departure and the price within 14 days,
    return the profit from the RM1 policy, which is to increase price for last 14 days.'''
    vFirstPeriod=data[1,data[0]>14]
    vSecondPeriod=data[1,data[0]<=14]
    demand1=min(inventory,np.sum(vFirstPeriod>=price1))
    demand2=min(inventory-demand1,np.sum(vSecondPeriod>=price2))
    return price1*demand1+price2*demand2-fixedCost
```

This function is similar to constant price except that there is a `price1` for before last 14 days, and another `price2` for the last 14 days. The code here breaks up the valuation to those before 14 days and those after 14 days. This is illustrated as below, using the same smaller data object for ease of understanding.

```
[82]: data=np.array([[3,1,8,9,5],[6,7,2,4,10]])
      print('data=\n',data)
      print('data[:,data[0]<=8] filters columns such that first row is no more than 8\n',data
      print('data[1,data[0]<=8] displays only row 1 of the above\n',data[1,data[0]<=8])

data=
 [[ 3  1  8  9  5]
 [ 6  7  2  4 10]]
data[:,data[0]<=8] filters columns such that first row is no more than 8
 [[ 3  1  8  5]
 [ 6  7  2 10]]
data[1,data[0]<=8] displays only row 1 of the above
 [ 6  7  2 10]
```

Exercise: Declare an array `np.array([[1,3,4,5],[2,1,3,7],[3,2,1,0]])`

- Filter for all columns such that row 0 is at least 3.
- Select row 2 such that row 0 is at least 3.
- Select column 3 such that column 0 is less than or equal to 2.

Answer:

```
[105]: a=np.array([[1,3,4,5],[2,1,3,7],[3,2,1,0]])
       print('Array a=\n',a)
       print('Filtering all columns such that row 0 is at least 3\n',a[:,a[0]>=3])
       print('Get row 2 such that row 0 is at least 3\n',a[2,a[0]>=3])
       print('Select column 3 such that column 0 is <= 2 (note that column is now 1-D so disp


Array a=
 [[1 3 4 5]
 [2 1 3 7]
 [3 2 1 0]]
Filtering all columns such that row 0 is at least 3
 [[3 4 5]
```

```
 [1 3 7]
 [2 1 0]]
Get row 2 such that row 0 is at least 3
 [2 1 0]
Select column 3 such that column 0 is <= 2 (note that column is now 1-D so displayed in row f
 [5 7]
```

RM 2 Policy: Increasing Price after Certain Quantity Sold

```
def simulateRM2(data,price1,price2,quantity):
    '''Given the simulated scenario from generateScenario() (stored in the data variable),
    as well as the initial price, the later price, and the quantity to sell before increasing
    return the profit from the RM2 policy, which is to increase price after selling a certain
    sold=0
    rev=0
    curPrice=price1
    for t,v in data.T:
        if sold==quantity:
            curPrice=price2
        if v>=curPrice:
            sold+=1
            rev+=curPrice
        if sold==inventory:
            break
    return rev-fixedCost
```

The commands used here are relatively simple.

- Assignments of data to variable.
- `for` loop to iterate through the columns of data (`data.T` transposes the array, so we are now iterating through the columns instead of the rows.) Specifically, `for t,v in data.T` goes through each column of data, which is an array of 2 numbers, and assign the first number to `t` and the second to `v`. This is because the in the output of `generateScenario()`, row 0 is buy time and row 1 is valuations.
- `if` statement to decide whether we have sold up to quantity yet, as well as whether the current customer would buy.
- `break` statement for exiting the loop when we have sold enough inventory. In general, `break` exists whatever loop we are in. (For nested loops, it only exits one layer of the loop.)

However, the logic is more involved than the other functions, as there are several `if` statements. In these cases, a useful technique for **disecting** code is to feed it a small set of parameters, and add print statements so we can watch it go through each step. Below is an illustration. **Look at the output first for clarity.**

```
[111]: def simulateRM2_verbose(data,price1,price2,quantity):
           print('Executing simulateRM2 with data=\n{0}\n price1={1} price2={2} quantity={3}\
           sold=0
           rev=0
           curPrice=price1
           print('Initialized sold={0}, rev={1}, curPrice={2}'.format(sold,rev,curPrice))
```

```
            for t,v in data.T:
                print('\t new customer came at t={0} with valuation v={1}, curPrice={2}'.forma
                if sold==quantity:
                    curPrice=price2
                    print('\t\t sold=={0}, Incrementing curPrice={1}'.format(quantity,curPrice
                if v>=curPrice:
                    sold+=1
                    rev+=curPrice
                    print('\t\t v>=curPrice. Updating sold={0}, rev={1}'.format(sold,rev))
                if sold==inventory:
                    print('\t\t sold==inventory. Exit loop')
                    break
            print('Returning rev={0} minus fixedCost={1}'.format(rev,fixedCost))
            return rev-fixedCost

        data=np.array([[180,178,150,140,130,129],[200,150,100,250,500,300]])
        quantity=2
        price1=200
        price2=300
        profit=simulateRM2_verbose(data,price1,price2,quantity)
        print('Final output:',profit)
```

```
Executing simulateRM2 with data=
[[180 178 150 140 130 129]
 [200 150 100 250 500 300]]
 price1=200 price2=300 quantity=2

Initialized sold=0, rev=0, curPrice=200
         new customer came at t=180 with valuation v=200, curPrice=200
                 v>=curPrice. Updating sold=1, rev=200
         new customer came at t=178 with valuation v=150, curPrice=200
         new customer came at t=150 with valuation v=100, curPrice=200
         new customer came at t=140 with valuation v=250, curPrice=200
                 v>=curPrice. Updating sold=2, rev=400
         new customer came at t=130 with valuation v=500, curPrice=200
                 sold==2, Incrementing curPrice=300
                 v>=curPrice. Updating sold=3, rev=700
         new customer came at t=129 with valuation v=300, curPrice=300
                 v>=curPrice. Updating sold=4, rev=1000
Returning rev=1000 minus fixedCost=20000
Final output: -19000
```

Exercise:

Modify the `simulateRM2` function to allow for a second price increase at another quantity. Specifically, write a function of the form

```
def simulateRM3(data,price1,price2,price3,quantity1,quantity1)
```

Such that the initial price is price1, but when `quantity1` # of tickets is sold, then the price changes to price2. Moreover, when `quantity2` # of tickets is sold, then the price changes again to price3.

Answer:

```
[112]: def simulateRM3_verbose(data,price1,price2,price3,quantity1,quantity2):
           print('Executing simulateRM3 with data=\n{0}\n price1={1} price2={2} price3={3} qu
           sold=0
           rev=0
           curPrice=price1
           print('Initialized sold={0}, rev={1}, curPrice={2}'.format(sold,rev,curPrice))
           for t,v in data.T:
               print('\t new customer came at t={0} with valuation v={1}, curPrice={2}'.forma
               if sold==quantity1:
                   curPrice=price2
                   print('\t\t sold=={0}, updating curPrice={1}'.format(quantity1,price2))
               elif sold==quantity2:
                   curPrice=price3
                   print('\t\t sold=={0}, updating curPrice={1}'.format(quantity2,price3))

               if v>=curPrice:
                   sold+=1
                   rev+=curPrice
                   print('\t\t v>=curPrice. Updating sold={0}, rev={1}'.format(sold,rev))

               if sold==inventory:
                   print('\t\t sold==inventory. Exit loop')
                   break
           print('Returning rev={0} minus fixedCost={1}'.format(rev,fixedCost))
           return rev-fixedCost

       data=np.array([[180,178,150,140,130,129],[200,150,100,250,500,300]])
       quantity1=2
       quantity2=3
       price1=100
       price2=200
       price3=400
       profit=simulateRM3_verbose(data,price1,price2,price3,quantity1,quantity2)
       print('Final output:',profit)
```

```
Executing simulateRM3 with data=
[[180 178 150 140 130 129]
 [200 150 100 250 500 300]]
 price1=100 price2=200 price3=400 quantity1=2 quantity2=3

Initialized sold=0, rev=0, curPrice=100
         new customer came at t=180 with valuation v=200, curPrice=100
                 v>=curPrice. Updating sold=1, rev=100
         new customer came at t=178 with valuation v=150, curPrice=100
                 v>=curPrice. Updating sold=2, rev=200
         new customer came at t=150 with valuation v=100, curPrice=100
                 sold==2, updating curPrice=200
         new customer came at t=140 with valuation v=250, curPrice=200
                 sold==2, updating curPrice=200
                 v>=curPrice. Updating sold=3, rev=400
         new customer came at t=130 with valuation v=500, curPrice=200
                 sold==3, updating curPrice=400
```

```
                v>=curPrice. Updating sold=4, rev=800
        new customer came at t=129 with valuation v=300, curPrice=400
Returning rev=800 minus fixedCost=20000
Final output: -19200
```

Note that the line `sold==2, updating curPrice=200` occurred twice in the above output. This is because after the first update, the customer did not buy, so the number sold remained at 2 in the next loop. We can eliminate the redundant checking by moving the if statements that check for `if sold==quantity...` into the if statement `if v>=curPrice`, after we increment `sold+=1`, so that we only check the quantity sold after it changes. This does not change the output of the code but makes it more efficient.

Optimization by Enumeration

After completing the `generateScenario`, `simulatePrice`, `simulateRM1` and `simulateRM2` functions, the provided code in Lab 2 uses these functions to optimize for the best parameters under each pricing policy.

Optimizing the Constant Price Policy

```
def analyzePrice():
    prices=range(150,300,1)
    np.random.seed(0)
    dataSet=[generateScenario() for t in range(100)]
    values=[np.average([simulatePrice(data,price) for data in dataSet]) for price in prices]

    plt.plot(prices,values)
    plt.title('Revenue from constant price policy')
    plt.xlabel('Price')
    plt.ylabel('Revenue')
    plt.show()

    index=values.index(max(values))
    print ('Best constant price = {0}, best profit={1}'.format(prices[index],values[index]))


analyzePrice()
```

The above code looks at every price in `range(150,300,1)`, which is `[150,151,..., 299]` and compute the average profit in 100 generated scenarios. To save time, the code first generates all 100 scenarios, and use the same scenarios for all prices. It then plots the profit (averaged from 100 scenarios) against the price, and return the best price.

Setting the random seed

The line `np.random.seed(0)` makes sure that every time the code is run, the random scenario generated are the same. This is not necessary but it is helpful as it ensures that the constant price policy is getting the same scenarios as the RM1 and RM2 policies, which makes the comparison fair. Instead of `0`, you can set the seed to any number you like. Consider the following example.

```
[114]: from scipy.stats import norm
       X=norm(3,1)
       print('Setting seed to 0')
       np.random.seed(0)
       print('First random number',X.rvs())
       print('Second random number',X.rvs())
       print('third random number',X.rvs())
       print('Resetting seed to 0')
       np.random.seed(0)
       print('fourth random number',X.rvs())
       print('fifth random number',X.rvs())
```

```
Setting seed to 0
First random number 4.76405234597
Second random number 3.40015720837
third random number 3.97873798411
Resetting seed to 0
fourth random number 4.76405234597
fifth random number 3.40015720837
```

Notice how after setting the seed again, we get the same sequence of numbers as before.
Exercise:

- Generate 5 random uniformly random numbers between 10 and 15 (using `rvs(size=5)`). Then generate another 5 uniformly random numbers. Notice the numbers are different.
- Now do the same exercise, but set the seed to 3 before generating each one. Notice now the numbers are the same.

```
[115]: from scipy.stats import uniform
       X=uniform(10,5)
       print('First set of numbers',X.rvs(size=5))
       print('Second set of numbers',X.rvs(size=5))
       np.random.seed(3)
       print('Third set of numbers',X.rvs(size=5))
       np.random.seed(3)
       print('Fourth set of numbers',X.rvs(size=5))
```

```
First set of numbers [ 13.01381688  12.72441591  12.118274    13.22947057  12.18793606]
Second set of numbers [ 14.458865    14.8183138   11.91720759  13.95862519  12.6444746 ]
Third set of numbers [ 12.75398951  13.54073911  11.45452369  12.55413803  14.46473477]
Fourth set of numbers [ 12.75398951  13.54073911  11.45452369  12.55413803  14.46473477]
```

List comprehension

The following code uses a Python shortcut for creating lists, called *list comprehension*.

```
dataSet=[generateScenario() for t in range(100)]
values=[np.average([simulatePrice(data,price) for data in dataSet]) for price in prices]
```

Essentially, the first line produces the same result as

```
dataSet=[]
for t in range(100):
    dataSet.append(generateScenario())
```

Instead of declaring a `for` loop and adding items to the list one by one, the `dataSet=[generateScenario() for t in range(100)]` does everything in one line.

The second line is equivalent to

```
values=[]
for price in prices:
    profitSamples=[]
    for data in dataSet:
        profitSamples.append(simulatePrice(data,price))
    averageProfit=np.average(profitSamples)
    values.append(averageProfit)
```

List comprehension does the looping and list creation together in an intuitive way, while having the advantage of saving lines of code and speeding up the internal processing. Below are some further examples of list comprehension. Notice that list comprehension also allows for multiple loops.

```
[128]: print([2*i for i in [3,4,5]])
       print([-i for i in range(10)])
       print([i*100+j for i in range(1,4) for j in range(5) ])
```

```
[6, 8, 10]
[0, -1, -2, -3, -4, -5, -6, -7, -8, -9]
[100, 101, 102, 103, 104, 200, 201, 202, 203, 204, 300, 301, 302, 303, 304]
```

Exercise:

- Use list comprehension to create a list with values 1×1, 2×2 all the way up to 10×10.
- Use `np.average` to compute the average value of this list.
- Use list comprehension to create a list of size 10, each with 5 draws of a `norm(1,1)` random variable.

Answers:

```
[129]: print([i*i for i in range(1,11)])
       print(np.average([i*i for i in range(1,11)]))
       print([norm(1,1).rvs(size=5) for i in range(5)])
```

```
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
38.5
[array([ 1.47145428,  1.23293499,  1.8533351 ,  0.74755619,  1.18251382]), array([ 0.88428064
```

Note: we could have created a 10 by 5 array of random numbers by supplying the tuple `(10,5)` to size directly. As below.

```
[130]: print(norm(1,1).rvs(size=(10,5)))
```

```
[[ 0.9058414   1.31601162  0.53530541 -0.55610702  1.70821457]
 [ 0.7627285   2.30780347  0.92835475 -1.27907707  2.52080411]
 [ 1.69483058  1.2932445   0.17217511  0.68831159  2.82097863]
 [-0.81253704  1.65228156  1.51566148 -0.02121666  0.55107647]
 [ 2.37849537 -0.02035751 -1.68411037  1.02830028  1.71088548]
 [ 1.19504572  1.51882028  0.21597799  2.04989656  0.49436353]
 [ 1.53321922  1.42745909  1.22394039  0.52049339  0.05611494]
 [ 0.217179    1.37070428  0.14144409  1.49824704  1.63040281]
 [-0.36760905  1.30175674 -0.22001177 -1.47660281  1.37394942]
 [-2.05140943  1.1090795   1.44098817 -0.34603852 -1.7309097 ]]
```

Searching through a list

The code

```
index=values.index(max(values))
print ('Best constant price = {0}, best profit={1}'.format(prices[index],values[index]))
```

finds the first index of the list `values` with the maximum value, as well as the corresponding price. Below is further illustration.

```
[131]: l=[3,4,5,2,5]
       print('list l=',l)
       print('max(l)=',max(l))
       print('l.index(3)=',l.index(3))
       print('l.index(4)=',l.index(4))
       print('l.index(5)=',l.index(5))
       print('l.index(2)=',l.index(2))

list l= [3, 4, 5, 2, 5]
max(l)= 5
l.index(3)= 0
l.index(4)= 1
l.index(5)= 2
l.index(2)= 3
```

Exercise:

- Define a list `a=[3,9,8,1,2,4]`. Use the functions `max` `min` and `a.index` to find the max value, minimum value, and the indices of the min and max values. Also find the index of the number 8.

Answer:

```
[133]: a=[3,9,8,1,2,4]
       print('max(a)=',max(a))
       print('min(a)=',min(a))
       print('a.index(max(a))=',a.index(max(a)))
       print('a.index(min(a))=',a.index(min(a)))
       print('a.index(8)=',a.index(8))
```

```
max(a)= 9
min(a)= 1
a.index(max(a))= 1
a.index(min(a))= 3
a.index(8)= 2
```

Optimizing the RM1 and RM2 policies

The code for finding the best parameters for the RM1 and RM2 policies are similar.

```
def analyzeRM1():
    bestRev=0
    bestPrice1=0
    bestPrice2=0
    np.random.seed(0)
    dataSet=[generateScenario() for t in range(100)]
    for price1 in range(150,500,5):
        for price2 in range(price1,500,5):
            rev=np.average([simulateRM1(data,price1,price2) for data in dataSet])
            if rev>bestRev:
                bestRev=rev
                bestPrice1=price1
                bestPrice2=price2
    print ('Best RM1 policy: price1={0} price2={1}, profit={2}'.format(bestPrice1,bestPrice2,

def analyzeRM2():
    bestRev=0
    np.random.seed(0)
    dataSet=[generateScenario() for t in range(100)]
    for price1 in range(225,246,5):
        for price2 in range(290,331,5):
            for quantity in range(1,100):
                rev=np.average([simulateRM2(data,price1,price2,quantity) for data in dataSet]
                if rev>bestRev:
                    bestRev=rev
                    bestPrice1=price1
                    bestPrice2=price2
                    bestQuantity=quantity

    print ('Best RM2 policy: price1={0} price2={1}, quantity to sell before raising price={2}
analyzeRM2()
```

As with the constant price policy, we use list comprehension to generate a list called `dataSet` containing 100 sets of scenarios, and we set the seed to 0 to ensure that the same 100 scenarios is generated each time. Each function then uses multiple loops to search through a set of prices. In `analyzeRM1` we let `price1` search over `[150,155,160,...,495]` and `price2` increase from `price1` up to `495`. We increment in unit of 5 because the problem assumed that the price will be a multiple of 5. You could have searched over a larger range but the optimal answer is contained in this range. As in the lab 2 solution, the optimal setting is `price1=235`, `price2=310`.

In `analyzeRM2`, we let `price1` and `price2` vary around the optimal price in `analyzeRM1`, while having `quantity` go from 1 to 99. The reason we narrow to this range is that the optimal

turned out to be around this range (you could have searched over a larger range but the code would take longer to run).

In the inner loop, whenever we find a setting of parameters resulting in the profit averaged across the 100 scenarios, (variable `rev`) being larger than the best found so far (variable `bestRev`), then we update `bestRev` as well as store the best set of parameters. As discussed in session 1, this strategy for optimization (trying every possibility) is called enumeration. In order for it to run in a reasonable time, we have to constrain the set of options to try, which we did here by making `analyzeRM2` search around the optimum of `analyzeRM1`.