Microsoft

# Data Science Essentials

Lab 3 – Simulation

## Overview

In this lab, you will learn how to create, run and interpret simulations using R or Python. Simulation is widely used in cases where estimates are required from complex distributions of values or a hierarchy of distributions.

In this lab you will estimate the range of expected profitability for a lemonade stand. The profitability of the lemonade stand depends on the number of customers arriving, the profit from the drinks they order, and the tips the customer may or may not choose to leave. The distribution of possible profits is thus, the joint distribution of customer arrivals, items ordered, and tips. In practice, such a complex distribution cannot be analyzed except using simulation.

## What You'll Need

To complete this lab, you will need the following:

- The files for this lab
- An R or Python development environment. The lab steps assume you are using R Studio (for R) or Spyder (for Python).

**Note**: To set up the required environment for the lab, follow the instructions in the Setup Guide for this course.

## Computing Random Variables

Both R and Python offer comprehensive support for computing random variables from probability distributions. There is support in both R and Python for a large number of 'named' distributions; e.g. Normal, Poisson, Beta. Additionally, you can define your own functions to compute specialized probability distributions. In this exercise, you will compute and evaluate named probability distributions using R or Python.

### Compute Random Variables with R

**Note**: If you prefer to work with Python, skip this procedure and complete the next procedure, *Compute Random Variables with Python*.

In this procedure, you will compute a random set of values representing the number of customer arrivals per day at the lemonade stand. This calculation assumes that the mean number of arrivals per day is 600, with a standard deviation of 30. In practice, these parameters would be known from past history.

The Poisson distribution is often used to model arrival rates. Arrival rates can be any event that occurs randomly, such as, the number of people visiting a web site, the number of cars entering a highway, or the number of customers arriving at the lemonade stand. In this procedure you will compare results from generating arrivals from a Poisson and Normal distribution. Keeping in mind the Central Limit Theory, with a large mean and large number of realizations (computed random values) we expect the differences to be minimal.

1. Start RStudio, and open **restsim.R** from the lab folder for this module**.**
2. Verify that the code for the **sim.normal** function in the code editor looks like the following:

```
sim.normal <- function(num, mean = 600, sd = 30){
  ## Simulate from a Normal distribution
  dist = rnorm(num, mean, sd)
  titl = paste('Normal distribution with ', as.character(num), '
values')
  dist.summary(dist, titl)
  print('Emperical 95% CIs')
  print(comp.ci(dist))
  print(' ')
}
```

**Note**: This code uses the R **rnom()** function to generate random values drawn from a Normal distribution. A summary of the generated values along with confidence intervals are then computed. The default values of mean = 600 and standard deviation = 30 correspond to the distribution of expected arrivals of customers at the lemonade stand.

3. Look in the code editor window of RStudio and find the two functions called by the **sim.normal** function; **comp.ci** and **dist.summary**, which should look like this:

```
comp.ci <- function(vec, quantile = 0.05){
  ## Compute the upper and lower emperical quantiles
  lower <- quantile/2.0
  upper <- 1.0 - lower
  c(quantile(vec, probs = lower, na.rm = TRUE),
    quantile(vec, probs = upper, na.rm = TRUE))
}

dist.summary <- function(dist, name, num.bins = 120){
  ## function to plot and print a summary
  ## of the distribution
  maxm <- max(dist)
  minm <- min(dist)
  bw <- (maxm - minm)/num.bins
  breaks <- seq(minm - bw/2, maxm + bw/2, by = bw)
  hist(dist, col = 'blue', breaks = breaks, xlab = name,
       main = paste('Distribution of ', name))

  std <- round(sd(dist), digits = 2)
  print(paste('Summary of', name, '; with std = ', std))
  print(summary(dist))
```

```
}
```

**Note**: The **comp.ci** function computes the two-sided empirical confidence intervals or quantiles for the input values. The **dist.summary** function plots a histogram and returns summary statistics for the input values.

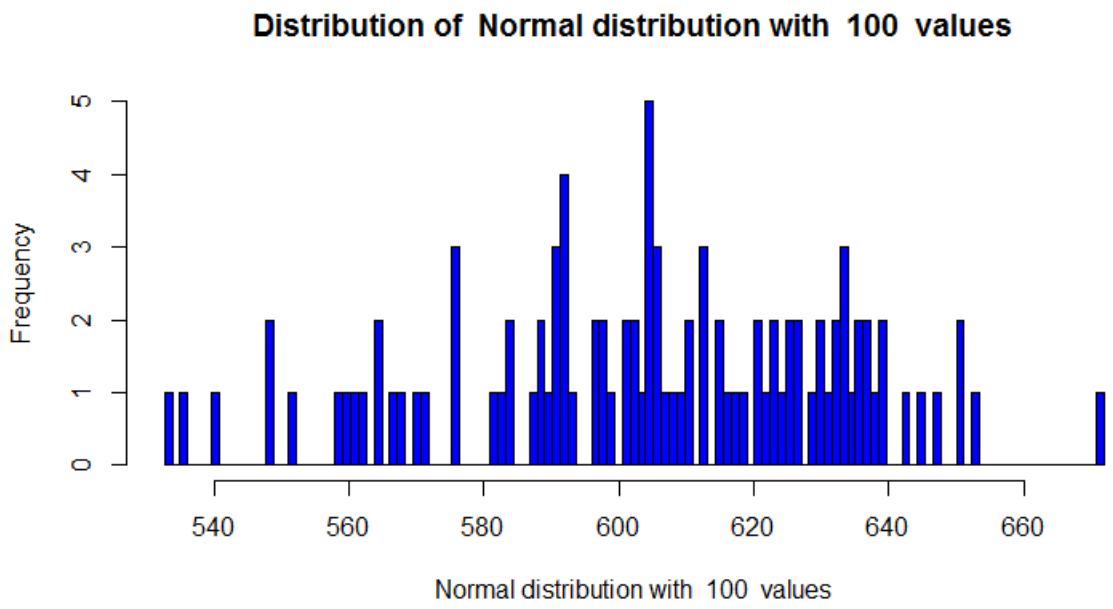
4. Click the **Source** icon in the upper right above the code editor window to add a reference to the **restsim.R** script in the Console window.
5. In the Console window, enter the following commands to run the **sim.normal** function for 100, 1,000, 10,000, and 100,000 values:

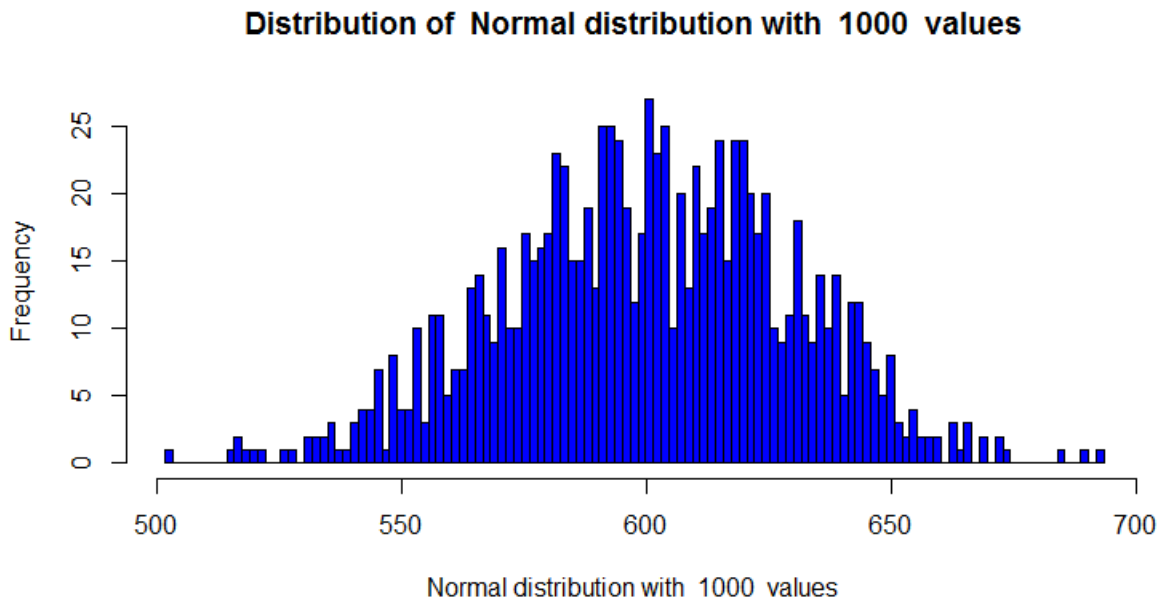
```
nums <- c(100, 1000, 10000, 100000)
lapply(nums, sim.normal)
```

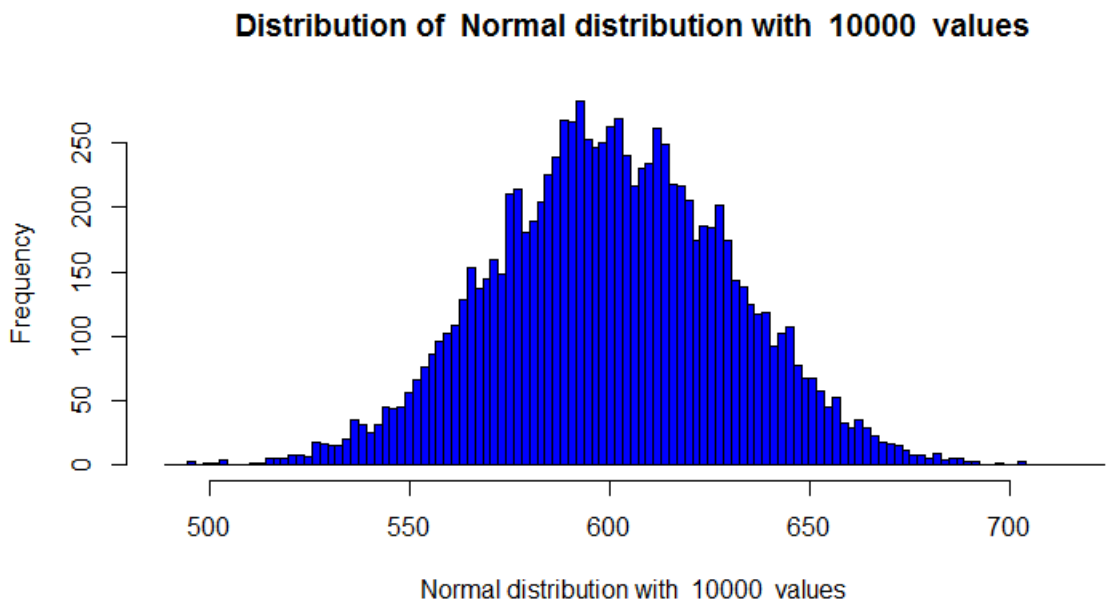
6. Review the printed output, which should look like this:

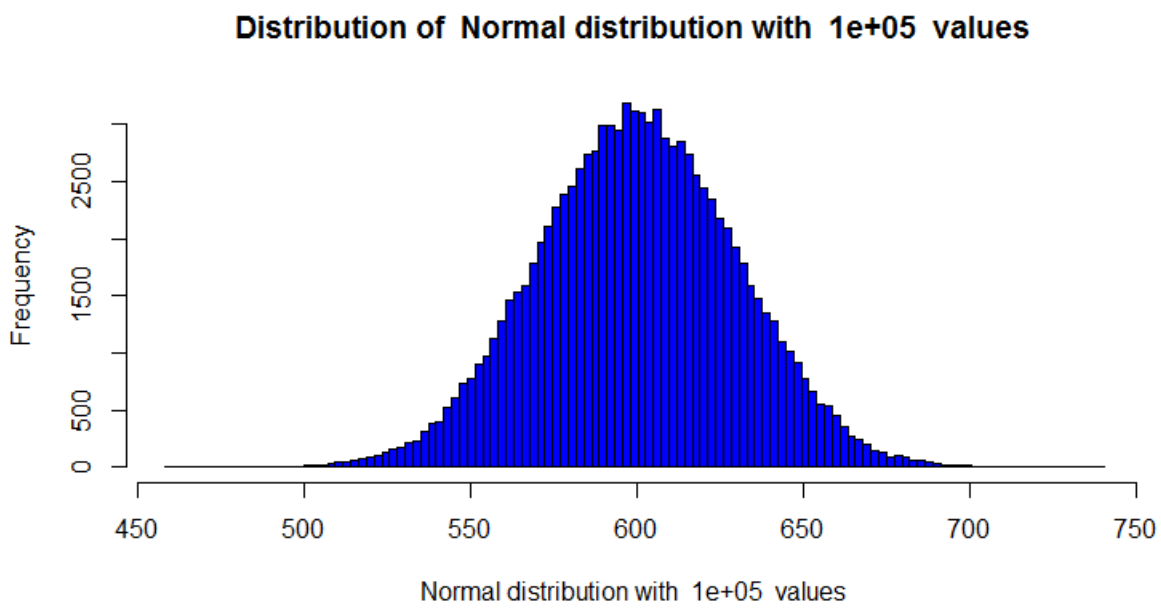
```
"Summary of Normal distribution with 100 values; with std = 28.78"
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  533.2   588.3   605.4   604.2   626.3   671.3
"Empirical 95% CIs"
    2.5%    97.5%
543.9033 650.3363

"Summary of Normal distribution with 1000 values; with std = 29.57"
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  502.2   580.0   601.0   599.9   620.5   692.6
"Empirical 95% CIs"
    2.5%    97.5%
542.3593 653.7556

"Summary of Normal distribution with 10000 values; with std = 29.93"
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  489.5   580.2   599.9   600.3   620.5   722.3
"Empirical 95% CIs"
    2.5%    97.5%
541.6681 658.8452

"Summary of Normal distribution with 1e+05 values; with std = 30"
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  459.3   579.7   600.0   600.0   620.4   739.1
"Empirical 95% CIs"
    2.5%    97.5%
541.6143 658.3349
```

7. Review the plots generated by the code, which should look like this (you will need to click the **Previous Plot** and **Next Plot** buttons in the **Plots** pane to view them all):


Distribution of Normal distribution with 100 values
Frequency
0
1
2
3
4
5
540
560
580
600
620
640
660
Normal distribution with 100 values


Distribution of Normal distribution with 1000 values
Frequency
0
5
10
15
20
25
500
550
600
650
700
Normal distribution with 1000 values




8. Note the following:
    - The median and mean converge to the theoretical values as the number of realizations (computed values) increases from 100 to 100,000. Likewise, the confidence intervals converge to their theoretical values.
    - The histogram of computed values comes to resemble the 'bell-shaped curve' of the theoretical Normal distribution as the number of realizations increases. Note that the histograms are affected by the quantization or binning of the values, which gives a somewhat bumpy appearance.

9. In the code editor, review the following code for the **sim.poisson** function. You will use this function to compare the results obtained simulating from a Normal distribution with a Poisson distribution:

```
sim.poisson <- function(num, mean = 600){
  ## Simulate from a Poisson distribution
  dist = rpois(num, mean)
  titl = paste('Poisson distribution with ', as.character(num), '
values')
  dist.summary(dist, titl)
  print('Emperical 95% CIs')
  print(comp.ci(dist))
  NULL
}
```

**Note**: The operation of this code is nearly identical to the **sim.normal** function you previously used. The **rpois** function has been substituted for the **rnorm** function.

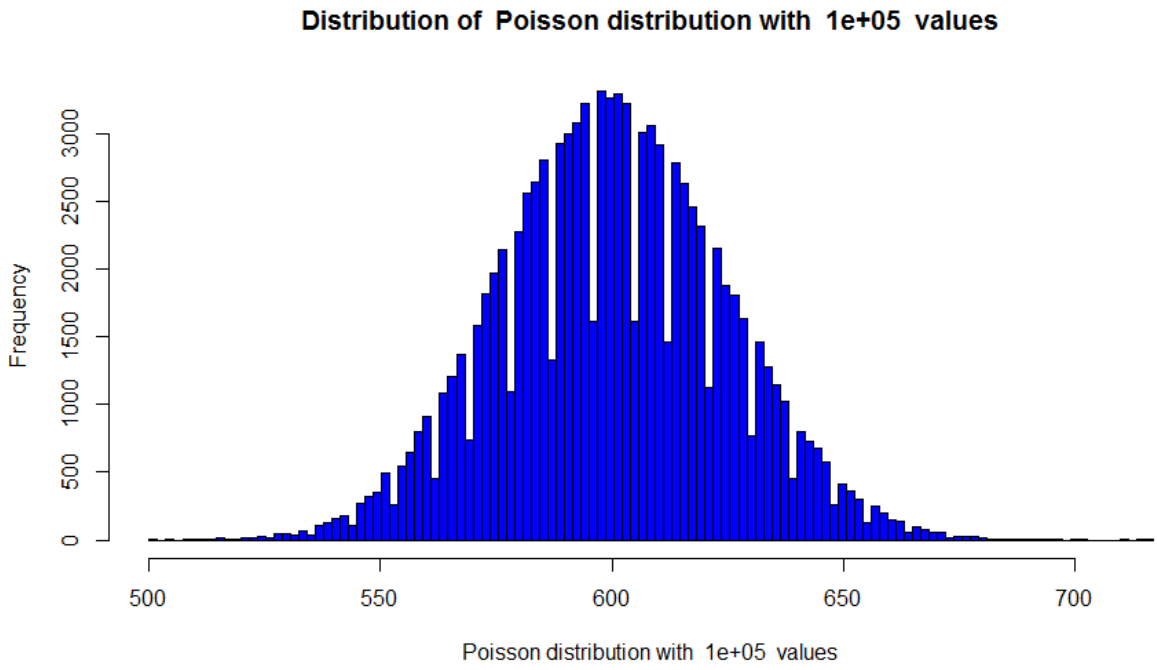
10. Run the **sim.poisson** function for 100,000 values by typing the following line of code in the RStudio console window:

```
sim.poisson(100000)
```

11. Review the printed and plotted output, which should look like the following:

```
"Summary of Poisson distribution with 1e+05 values; with std = 24.47"
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
    501     583     600     600     616     716
"Empirical 95% CIs"
 2.5% 97.5%
  553   648
```

12.




13. Examine these results for the Poisson distribution, comparing them to those for the Normal distribution, and note the following:

- The mean and median are at the theoretical values for the Normal distribution.
- The 95% two-sided confidence intervals differ only slightly from those for the Normal distribution.
- The values generated from a Poisson distribution are integers, which are reflected in the integer values for all the summary statistics. Further, this property leads to the uneven binning seen in the histogram.
- Despite the uneven binning, the general shape of the histogram is nearly identical to that for the Normal distribution.
- Overall it is safe to conclude that for the large value of the mean for the number of customer arrivals, there is no substantial difference between the Normal and Poisson distributions.

14. Leave your RStudio session running and proceed to the next exercise.

## Compute Random Variables with Python

**Note**: If you prefer to work with R, skip this procedure and complete the preceding procedure, *Compute Random Variables with R*.

1. Start Spyder and open the restsim.py script file from the lab files folder for this module**.**
2. Ensure that the code for the **sim_normal** function appears in the code editor window as follows:

```
def sim_normal(nums, mean = 600, sd = 30):
    import numpy.random as nr
    for n in nums:
        dist = nr.normal(loc = mean, scale = sd, size = n)
        titl = 'Normal distribution with ' + str(n) + ' values'
        print('Summary for ' + str(n) + ' samples')
        print(dist_summary(dist, titl))
        print('Emperical 95% CIs')
        print(quantiles(dist, [2.5, 97.5]))
        print(' ')
    return('Done!')
```

**Note**: This code uses the **normal** function from the Python **numpy.random** library to generate random values drawn from a Normal distribution. Summary statistics are then computed. Two-sided confidence intervals are computed using the **percentile** function from the Python numpy library. The default values of mean = 600 and standard deviation = 30 correspond to the distribution of expected arrivals of customers at the lemonade stand.

3. Note that the **sim_normal** function calls the **dis_summary** function in the **restsim.py** script file, which looks like this:

```
def dist_summary(dist, names = 'dist_name'):
    import pandas as pd
    import matplotlib.pyplot as plt
    ser = pd.Series(dist)
    fig = plt.figure(1, figsize=(9, 6))
    ax = fig.gca()
    ser.hist(ax = ax, bins = 120)
    ax.set_title('Frequency distribution of ' + names)
    ax.set_ylabel('Frequency')
    plt.show()
    return(ser.describe())
```

**Note**: This function plots a histogram of the values using the Pandas **hist** method. The function then returns some summary statistics for the values.

4. Change the working directory of your IPython session to the lab files folder for this module by typing the following at the IPython console prompts, being sure to use the full path to the lab files folder for this module:

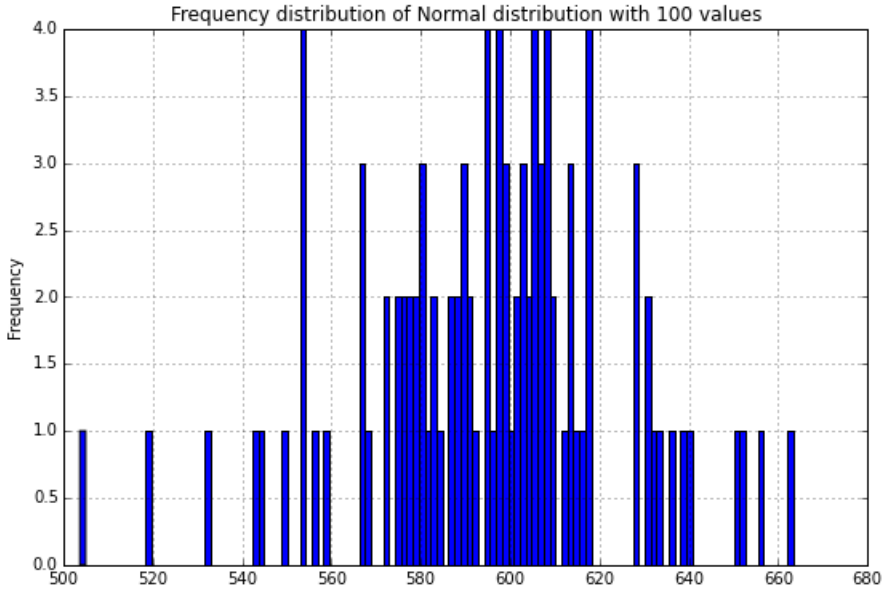
```
import os
os.chdir('YOUR-LAB-FOLDER-PATH')
```

5. Enter the following commands in the IPython console to load the code in the **restsim.py** file and then run the **sim_normal** function for 100, 1,000, 10,000, and 100,000 values:

```
import restsim as rs
nums = [100, 1000, 10000, 100000]
rs.sim_normal(nums)
```

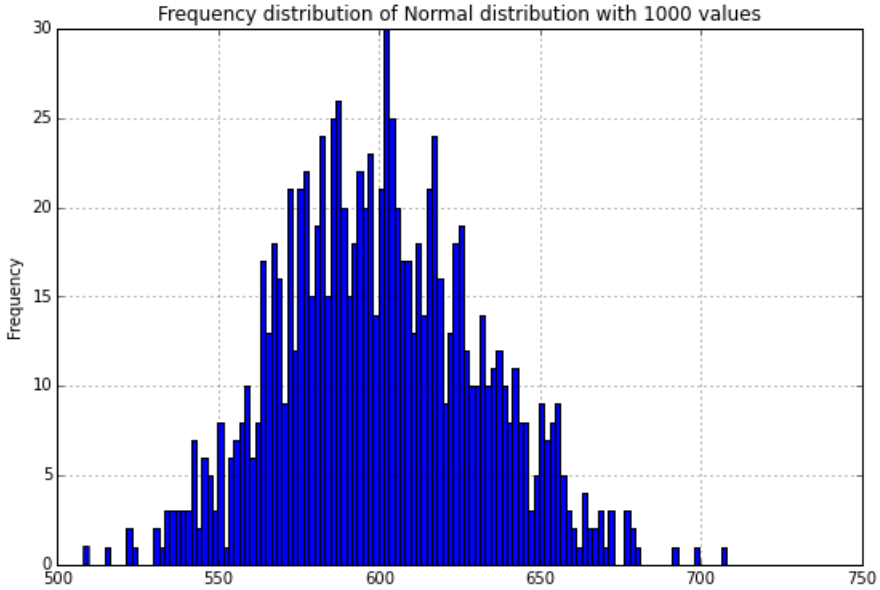
6. Review the printed and plotted output generated by the code, which should look similar to this:

```
Summary for 100 samples
```



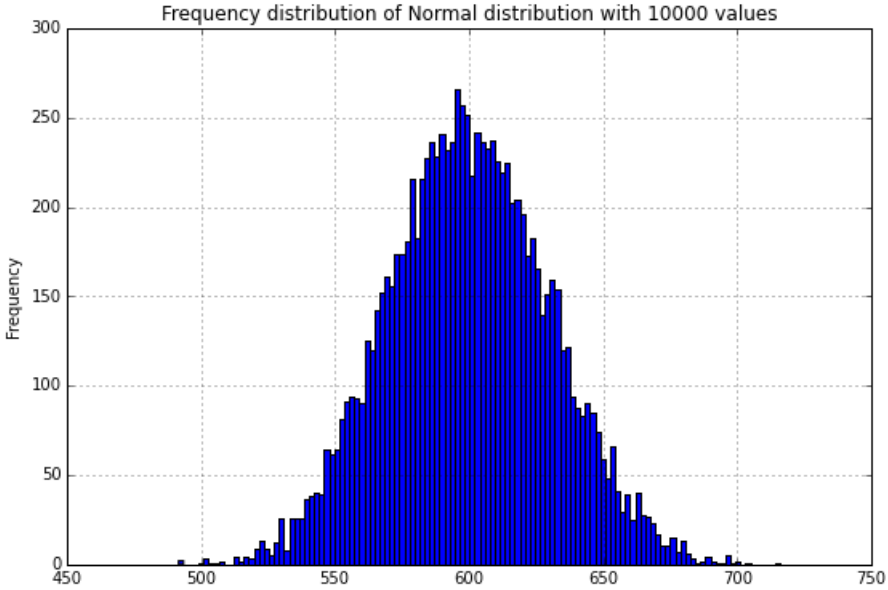

```
count    100.000000
mean     594.907361
std       28.361768
min      503.652063
25%      578.513507
50%      597.384307
75%      609.078271
max      663.390959
dtype: float64
Emperical 95% CIs
[ 537.59980488  651.54643004]
```

```
Summary for 1000 samples
```



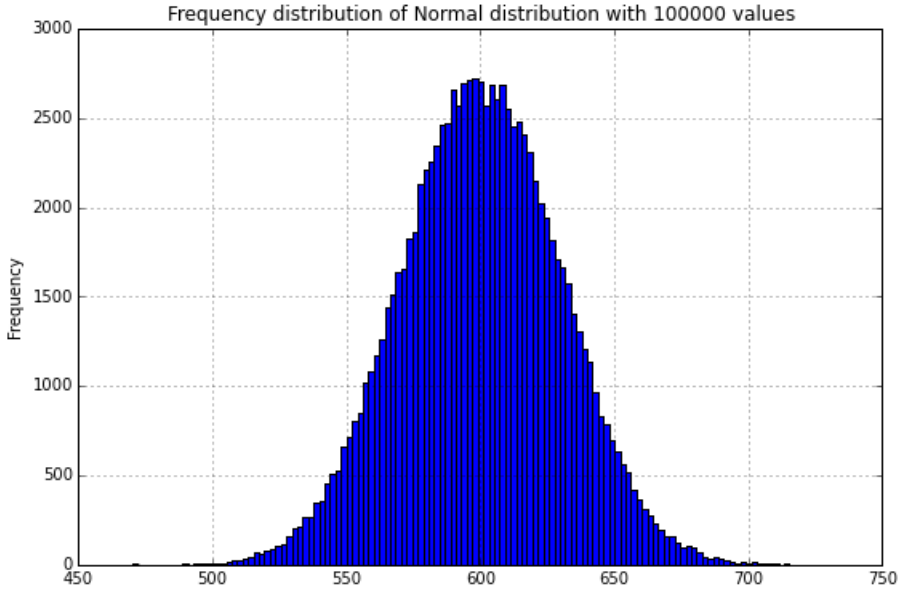

```
count    1000.000000
mean      600.140624
std        30.886523
min       508.105770
25%       578.212448
50%       599.056329
75%       621.125640
max       707.815185
dtype: float64
Emperical 95% CIs
[ 542.42715962  660.55401403]
```

```
Summary for 10000 samples
```




```
count    10000.000000
mean       599.628037
std         30.090041
min        491.713237
25%        578.933060
50%        599.044416
75%        619.823489
max        716.128577
dtype: float64
Emperical 95% CIs
[ 541.97762393  659.70216395]
```

```
Summary for 100000 samples
```




```
count    100000.000000
mean        600.034905
std          29.809940
min         470.458606
25%         579.892918
50%         599.966959
75%         620.183680
max         715.391581
dtype: float64
Emperical 95% CIs
[ 541.73092162  658.10404261]
```

7. Examine these results, noting the following:
    - The median and mean converge to the theoretical values as the number of realizations (computed values) increases from 100 to 100,000. Likewise, the confidence intervals converge to their theoretical values.
    - The histogram of computed values comes to resemble the 'bell-shaped curve' of the theoretical Normal distribution as the number of realizations increases. Note that the histograms are affected by the quantization or binning of the values, which gives a somewhat bumpy appearance.

8. In the code editor, review the following code for the sim_poisson function. You will use this function to compare the results obtained simulating from a Normal distribution with a Poisson distribution:

```
def sim_poisson(nums, mean = 600):
    import numpy as np
    import numpy.random as nr
    for n in nums:
        dist = nr.poisson(lam = mean, size = n)
        titl = 'Poisson distribution with ' + str(n) + ' values'
        print(dist_summary(dist, titl))
        print('Emperical 95% CIs')
        print(np.percentile(dist, [2.5, 97.5]))
```

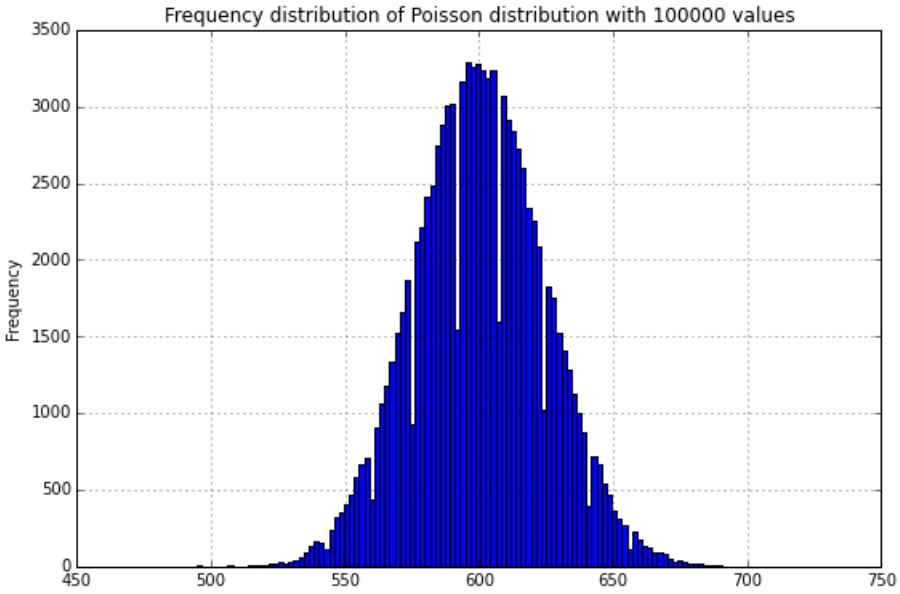
```
        print(' ')
    return('Done!')
```

Note: The operation of this code is nearly identical to the **sim_normal** function you previously used. The **poisson** function, from the **numpy.random** library, has been substituted for the **normal** function.

9. Enter the following command in the IPython console to compute and view results for the Poisson distribution for 100,000 values (be careful to include the square brackets, since the argument to the function must be an iteratetable object):

```
rs.sim_poisson([100000])
```

10. Review the results, which should look similar to this:




```
Icount    100000.000000n
mean         600.110050
std           24.414619
min          495.000000
25%          584.000000
50%          600.000000
75%          616.000000
max          721.000000
dtype: float64
Emperical 95% CIs
[ 553.  648.]
```

11. Examine these results for the Poisson distribution, and compare them to those for the Normal distribution. Note the following:
    - The mean and median (shown here as the 50% quantile) are at the theoretical values for the Normal distribution.
    - The 95% two-sided confidence intervals differ only slightly from those for the Normal distribution.

- The values generated from a Poisson distribution are integers, which are reflected in the integer values for all the summary statistics. Further, this property leads to the uneven binning seen in the histogram.
- Despite the uneven binning, the general shape of the histogram is nearly identical to that for the Normal distribution.
- Overall it is safe to conclude that for the large value of the mean for the number of customer arrivals there is no substantial difference between the Normal and Poisson distributions.

12. Leave your Spyder session running and proceed to the next exercise.

# Computing Specialized Random Variables

In the preceding exercise you computed random values from the Normal and Poisson distributions. There are many practical cases where a specialized distribution of values is needed. For the lemonade stand model, the profitability per order and the tip amount per customer visit are not standard distributions. In this case, you will use an R or Python function to compute these distributions. For the lemonade stand, assume that per order is 5 for 30% of customer visits, 3.5 for another 30% of customer visits, and 4 for the remaining 40% of visits. Assume that the tip is 0 for 50% of customer visits, 0.25 for 20% of visits, 1 for 20% of visits, and 2 for the remaining 10% of visits.

## Compute Specialized Random Variables with R

**Note**: If you prefer to work with Python, skip this procedure and complete the next procedure, *Compute Specialized Random Variables with Python*.

1. In the RStudio code editor window, find the **profits** function, which should look like the following:

```
profits <- function(num){
## Generates the profit from the uniform distribution
unif <- runif(num)
      ifelse(unif < 0.3, 5,
         ifelse(unif < 0.6, 3.5, 4))
}
```

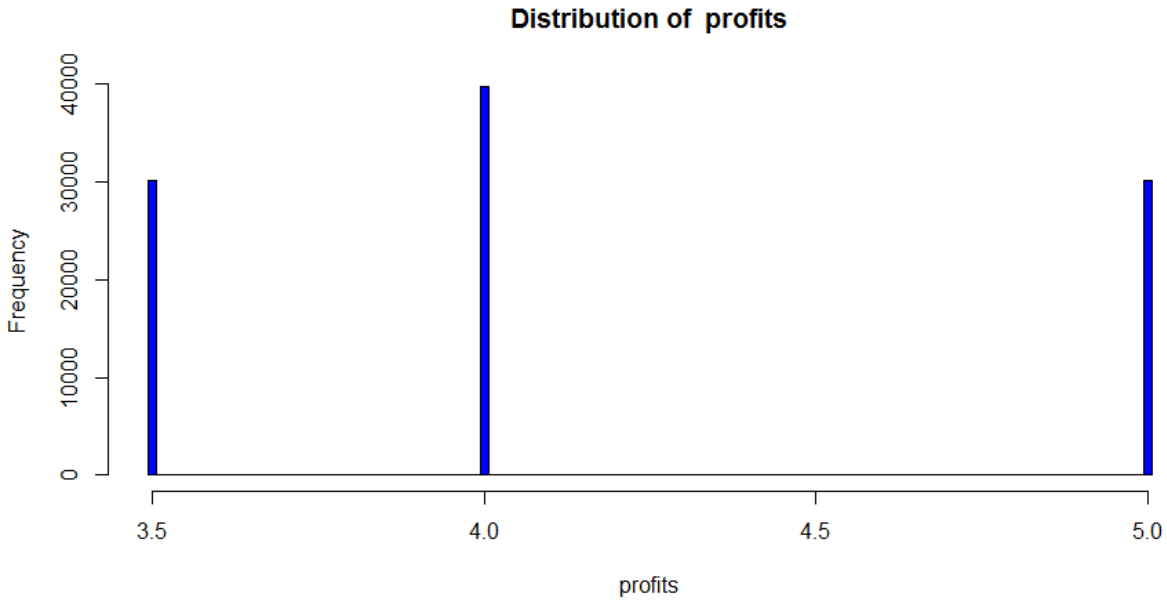
**Note**: This function using the R **runif** function to generate draws from a uniform distribution. Based on the values of the random draws, the profit is computed using the nested **ifelse** functions.

2. To test the **profits** function with 100,000 values, enter the following code in the console window:

```
prfts <- profits(100000)
dist.summary(prfts, 'profits')
```

3. Review the printed and plotted output, which should look like this:

```
"Summary of profits; with std = 0.6"
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
   3.50    3.50    4.00    4.15    5.00    5.00
```




**Note**: You can see that the distribution of profits per customer visit is as expected by looking at the frequencies of each profit value. Further, the median value is the most frequent profit level of 4.0.

4. In the code editor window, find the **tips** function, which should look like the following:

```
tips <- function(num){
  ## Generates the tips from the uniform distribution
  unif <- runif(num)
  ifelse(unif < 0.5, 0,
         ifelse(unif < 0.7, 0.25,
                ifelse(unif < 0.9, 1, 2)))
}
```

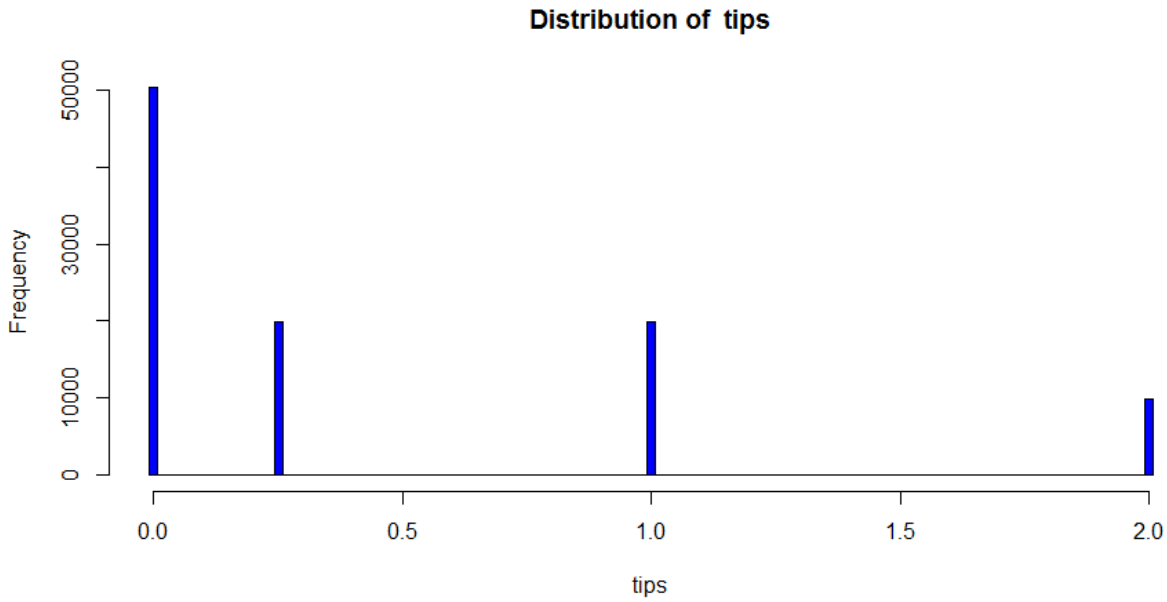
**Note**: This function is nearly identical to the **profits** function, except that there are two levels of nesting of the **ifelse** functions.

5. To test the **tips** function with 100,000 values, enter the following code in the console window:

```
tps <- tips(100000)
dist.summary(tps, 'tips')
```

6. Review the output, which should look like this:

```
"Summary of tips; with std = 0.64"
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
 0.0000  0.0000  0.0000  0.4477  1.0000  2.0000
```




**Note**: The distribution of tip values is as expected indicating the **tips** function operates correctly.

7. Keep your RStudio session running and proceed to the next exercise.

## Compute Specialized Random Variables with Python

**Note**: If you prefer to work with R, skip this procedure and complete the preceding procedure*, Compute Specialized Random Variables with R*.

1. Examine the code for the **gen_profits** function in the Spyder code editor window, which looks like this:

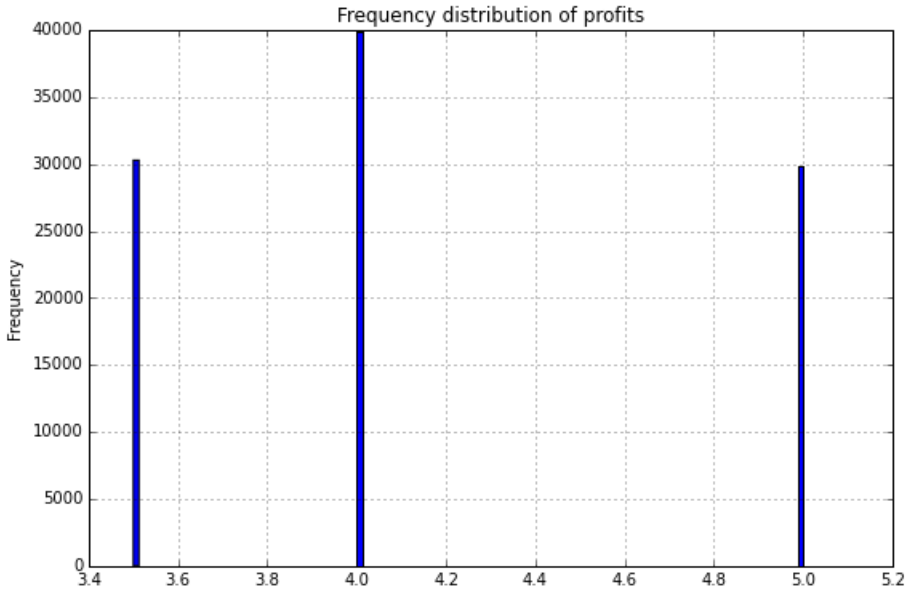
```
def gen_profits(num):
    import numpy.random as nr
    unif = nr.uniform(size = num)
    out = [5 if x < 0.3 else (3.5 if x < 0.6 else 4) for x in unif]
    return(out)
```

**Note**: This function generates random draws from a uniform distribution using the **uniform** function from the **numpy.random** library. Based on the values generated, the profit is computed using nested **if else** statements in the list comprehension.

2. To test this function with 100,000 values, enter the following code in the IPython console window:

```
prfts = rs.gen_profits(100000)
rs.dist_summary(prfts, 'profits')
```

3. Review the output, which should look similar to this:




```
Out[13]:
count    100000.000000
mean          4.149645
std           0.594035
min           3.500000
25%           3.500000
50%           4.000000
75%           5.000000
max           5.000000
dtype: float64
```

**Note**: You can see that the distribution of profits per customer visit is as expected by looking at the frequencies of each profit value. Further, the median value is the most frequent profit level of 4.0.

4. In the code editor window, find the **gen_tips** function, which should look like the following:

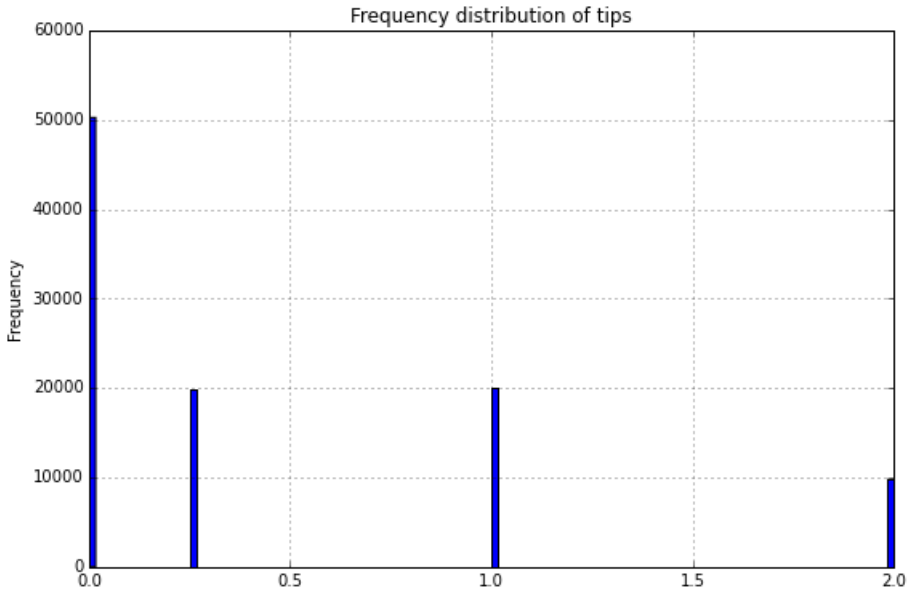
```
def gen_tips(num):
    import numpy.random as nr
    unif = nr.uniform(size = num)
    out = [0 if x < 0.5 else (0.25 if x < 0.7
      else (1.0 if x < 0.9 else 2.0)) for x in unif]
    return(out)
```

**Note**: This function is nearly identical to the **gen_profits** function, except that there are two levels of nesting of the if else statements in the list comprehension.

5. To test the **gen_tips** function with 100,000 values, enter the following lines of code in the IPython console:

```
tps = rs.gen_tips(100000)
rs.dist_summary(tps, 'tips')
```

6. Review the output, which should look similar to this:




```
Out[15]:
count    100000.000000
mean          0.449223
std           0.639851
min           0.000000
25%           0.000000
50%           0.250000
75%           1.000000
max           2.000000
dtype: float64
```

**Note**: The distribution of tip values is as expected, indicating the **gen_tips** function operates correctly.

7. Keep your Spyder session running and proceed to the next exercise.

## Simulating Lemonade Stand Income

In this exercise you will use Python or R to compute the distribution of total net daily income for the lemonade stand based on the number of visits, profit per sale, and tips. To compute total net income requires pulling together the pieces from the previous exercises and combining these results to form an overall model of lemonade stand income.

**Note**: The model represented by the code in this exercise makes some assumptions concerning independence of several of the random variables.

The frequency distribution of daily net income is modeled as p(net) = p(profit) + p(tips), where p(net) is the probability distribution of net income, p(profit) is the distribution of daily profit, and p(tips) is the distribution of daily tips. This model assumes that profit is independent of tip.

The frequency distribution of daily profit is the joint distribution of profit per customer arrival and customer arrivals per day; p(profit-visit, visits). Assuming independence between the visits and the

profit per visit, the join distribution is modeled as p(profit-visit, visits) = p(profit-visit)p(visits), where p(profit-visit) is the distribution of profits per visit, and p(visits) is the distribution of profits per day. Under the same assumption p(tips-visit, visits) = p(tips-visit)p(visits).

## Simulate lemonade Stand Income with R

**Note**: If you prefer to work with Python, skip this exercise and complete the next exercise, *Simulate Lemonade Stand Income with Python*.

1. Locate the **sim.lemonade** function in the code editor window of RStudio, which looks like the following:

```
sim.lemonade <- function(num, mean = 600, sd = 30, pois = FALSE){
  ## Simulate the profits and tips for
  ## a lemonade stand.

  ## number of customer arrivals
  if(pois){
    arrivals <- rpois(num, mean)
  } else {
   arrivals <- rnorm(num, mean, sd)
  }
  dist.summary(arrivals, 'customer arrivals per day')

  ## Compute distibution of average profit per arrival
  proft <- profits(num)
  dist.summary(proft, 'profit per arrival')

  ## Total profits are profit per arrival
  ## times number of arrivals.
  total.profit <- arrivals * proft
  dist.summary(total.profit, 'total profit per day')

  ## Compute distribution of average tips per arrival
  tps <- tips(num)
  dist.summary(tps, 'tips per arrival')

  ## Compute average tips per day
  total.tips <- arrivals * tps
  dist.summary(total.tips, 'total tips per day')

  ## Compute total profits plus total tips and normalize.
  total.take <- total.profit + total.tips
  dist.summary(total.take, 'total net per day')
}
```

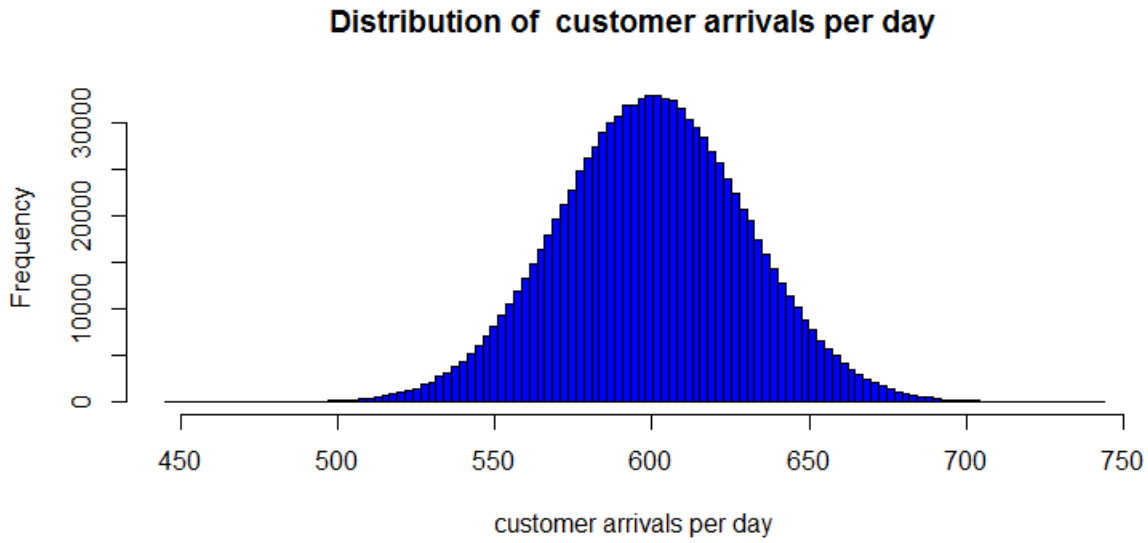
   Read this code, and take note of the comments, to understand the operations. Depending on the value of the **pois** argument, customer arrivals can be simulated from either a Normal or Poisson distribution.

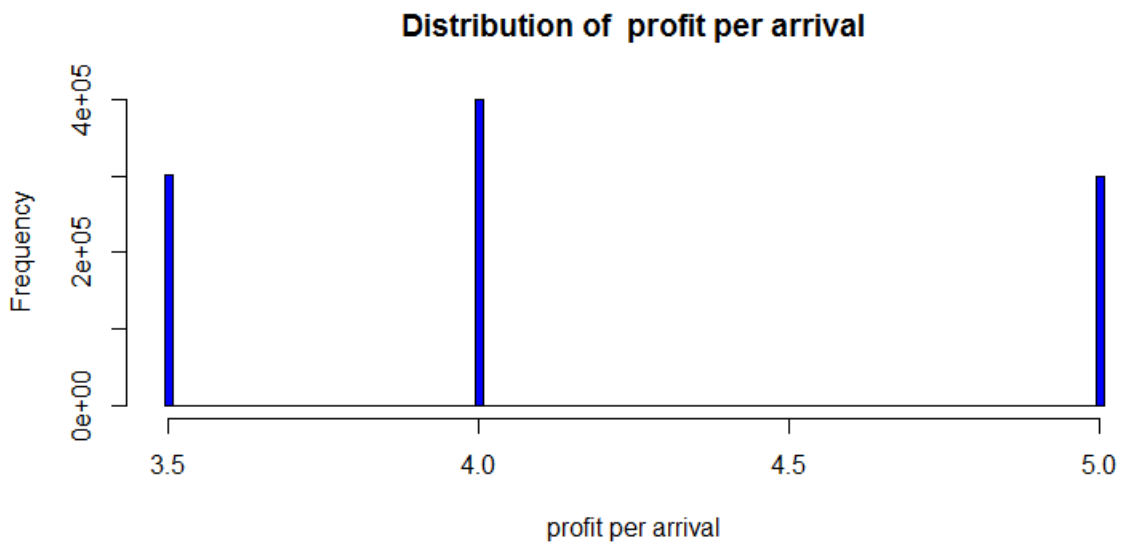
2. Enter the following code in the RStudio console to run the simulation with 100,000 values:

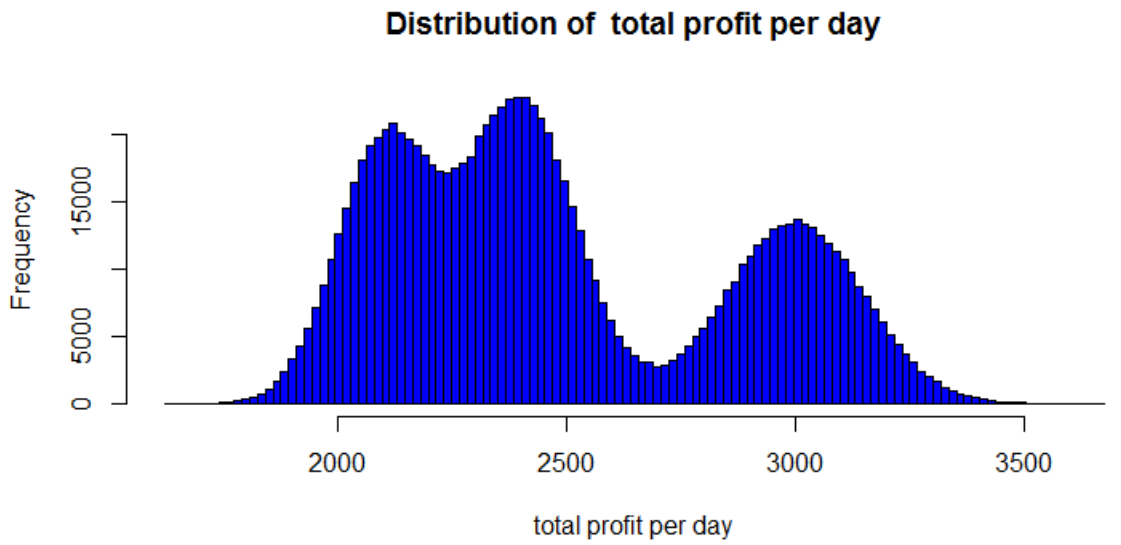
```
rs.sim_lemonade(100000)
```

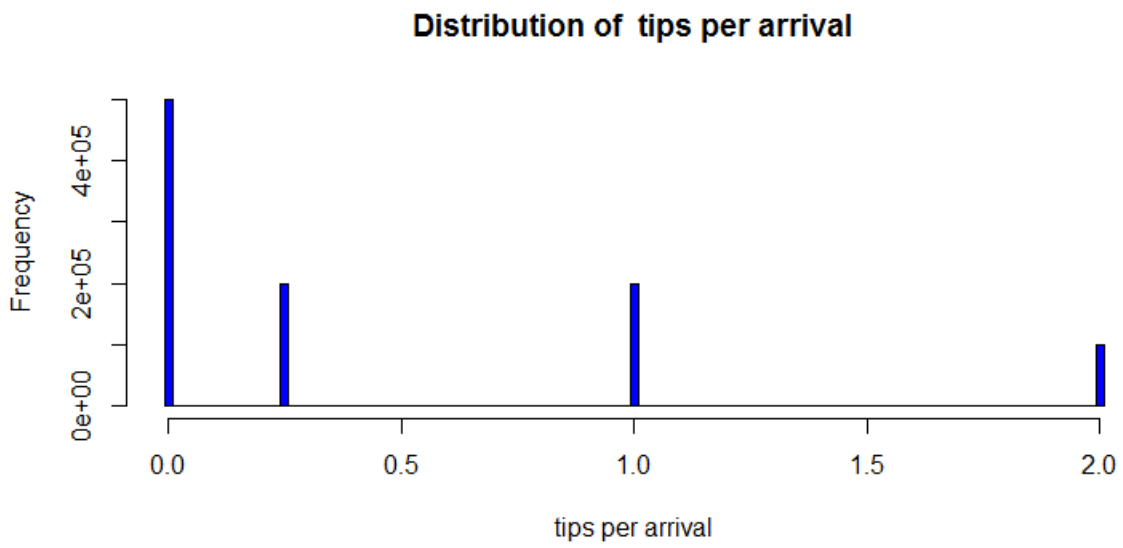
3. Review the printed and plotted output, which should look like this:

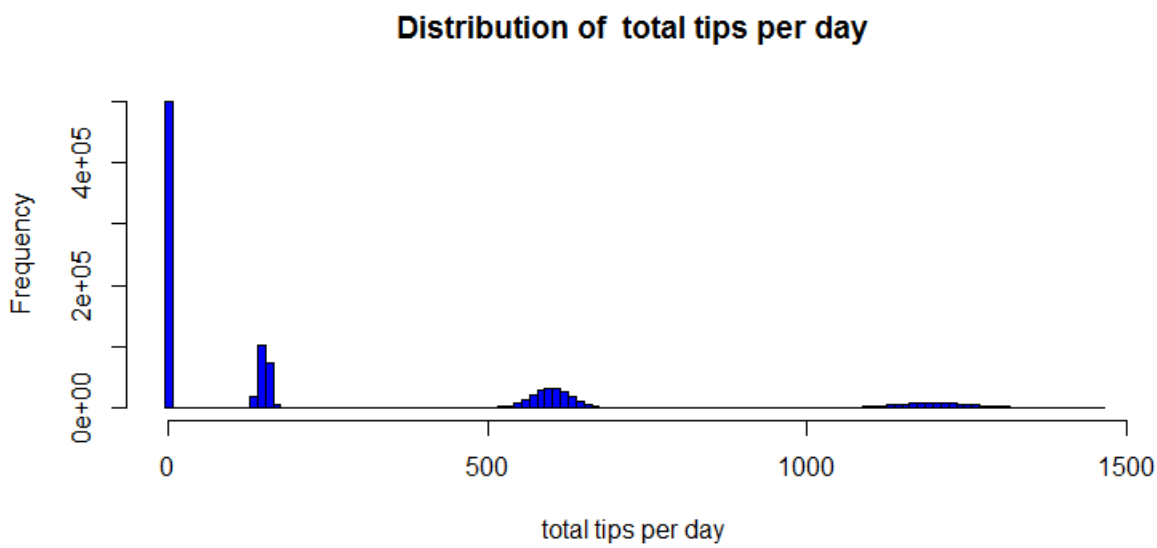
```
"Summary of customer arrivals per day ; with std =  29.94"
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  446.1   579.8   600.1   600.0   620.2   742.4

"Summary of profit per arrival ; with std =  0.59"
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
   3.50    3.50    4.00    4.15    5.00    5.00

"Summary of total profit per day ; with std =  377.69"
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
   1630    2183    2401    2490    2855    3666

"Summary of tips per arrival ; with std =  0.64"
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
 0.0     0.0000  0.0000  0.4499  1.0000  2.0000

"Summary of total tips per day ; with std =  384.98"
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
    0.0     0.0     0.0   269.9   579.8  1459.0

"Summary of total net per day ; with std =  542.81"
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
   1630    2325    2666    2760    3103    4990
```

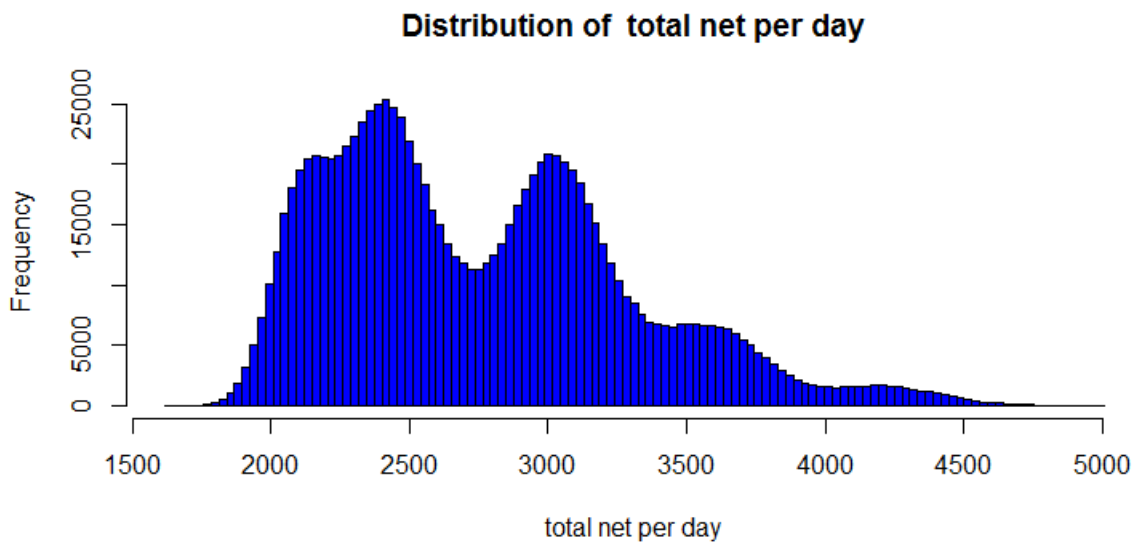



Distribution of profit per arrival
Frequency
0e+00
2e+05
4e+05
3.5
4.0
4.5
5.0
profit per arrival


Distribution of total profit per day
Frequency
0
5000
15000
2000
2500
3000
3500
total profit per day


Distribution of tips per arrival
Frequency
0e+00
2e+05
4e+05
0.0
0.5
1.0
1.5
2.0
tips per arrival


Distribution of total tips per day
Frequency
0e+00
2e+05
4e+05
0
500
1000
1500
total tips per day




Examine each of the plots and the corresponding summary statistics. Note the following:

- The Normal distribution of the customer arrivals is as expected with a mean and median of 600.
- The distribution of profits per arrival appears as was observed previously.
- The distribution of total profits per day is the vector product of arrivals per day and profits per arrival. Note, that this result is a complex distribution which would be difficult to handle except by simulation.
- The distribution of tips per arrival appears as was observed previously.
- The distribution of total tips per day, is the vector product of arrivals per day and tips per arrival. Again, this result is a complex distribution which would be difficult to handle except by simulation.
- The distribution of the final total net profit per day is the sum of the distribution of total profits per day and the distribution of total tips per day. This final distribution is quite complex with five peaks.

4. Enter the following command to run the simulation again, this time assuming a mean of 1200 customers per day with a standard deviation of 20:

```
sim.lemonade(100000, 1200, 20)
```

5. Review the resulting statistics and plots, comparing them with the simulation results for a 600 daily customer average.

**Note**: The distributions shown in this exercises are shown in terms of frequency of the values, not probability. The distribution would need to be normalized to transform frequency to probability values.

## Simulate Lemonade Stand Income with Python

**Note**: If you prefer to work with R, skip this procedure and complete the preceding procedure, *Simulate Lemonade Stand Income with R*.

1. Locate the **sim_lemonade** function in the code editor window of Spyder, which looks like the following:

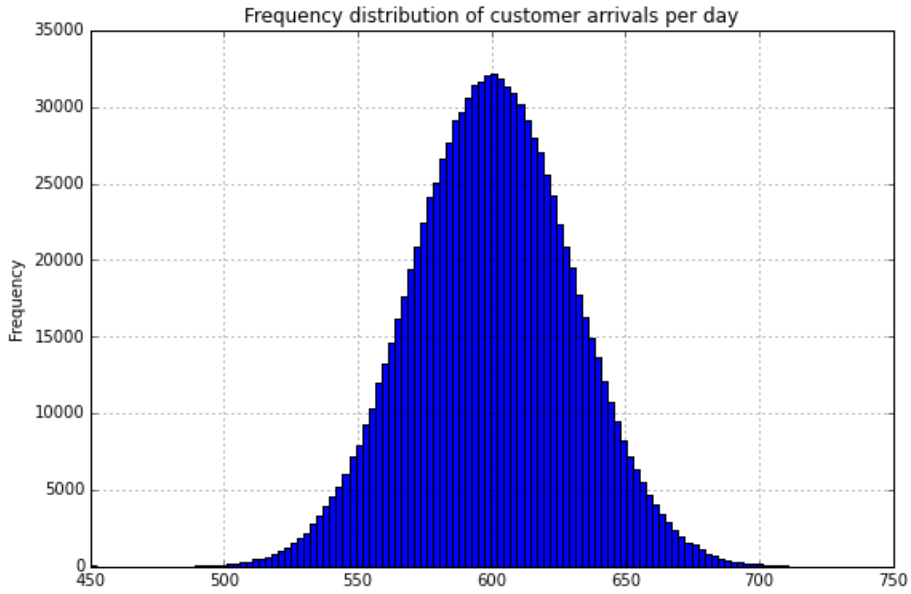
```
def sim_lemonade(num, mean = 600, sd = 30, pois = False):
  ## Simulate the profits and tips for
  ## a lemonade stand.
  import numpy.random as nr

  ## number of customer arrivals
  if pois:
    arrivals = nr.poisson(lam = mean, size = num)
  else:
   arrivals = nr.normal(loc = mean, scale = sd, size = num)

  print(dist_summary(arrivals, 'customer arrivals per day'))

  ## Compute distibution of average profit per arrival
  proft = gen_profits(num)
  print(dist_summary(proft, 'profit per arrival'))

  ## Total profits are profit per arrival
  ## times number of arrivals.
  total_profit = arrivals * proft
  print(dist_summary(total_profit, 'total profit per day'))

  ## Compute distribution of average tips per arrival
  tps = gen_tips(num)
  print(dist_summary(tps, 'tips per arrival'))

  ## Compute average tips per day
  total_tips = arrivals * tps
  print(dist_summary(total_tips, 'total tips per day'))

  ## Compute total profits plus total tips.
  total_take = total_profit + total_tips
  return(dist_summary(total_take, 'total net per day'))
```

   Read this code, and take note of the comments, to understand the operations. Depending on the value of the **pois** argument, customer arrivals can be simulated from either a Normal or Poisson distribution.

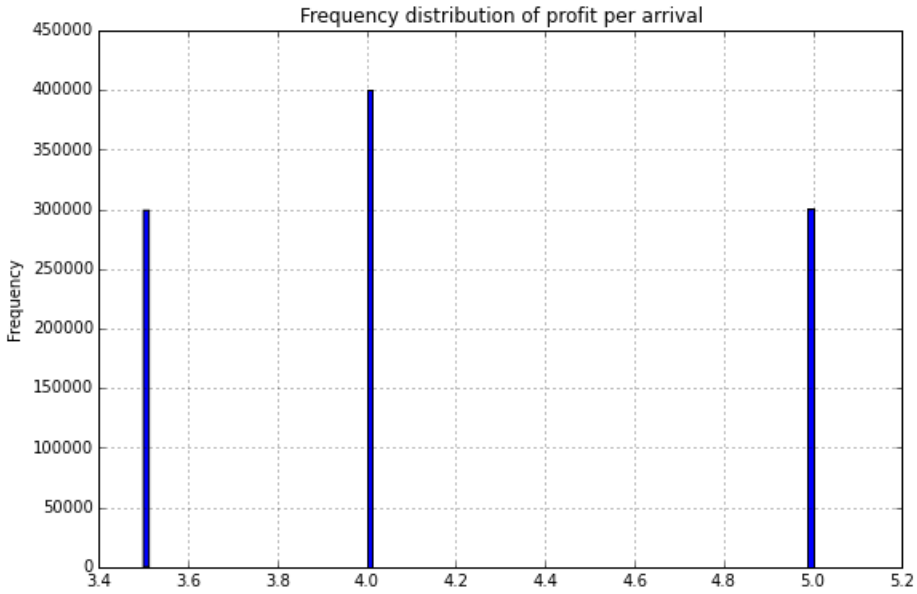
2. At the prompt, type the following line of code in the IPython console to run the simulation for 100,000 values:

```
rs.sim_lemonade(100000)
```

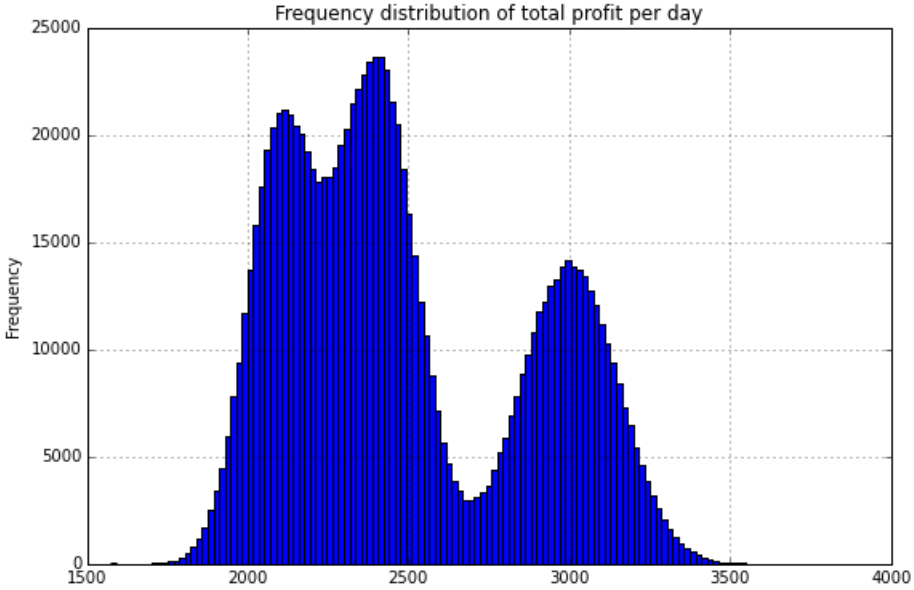
3. Review the output, which should look similar to this:



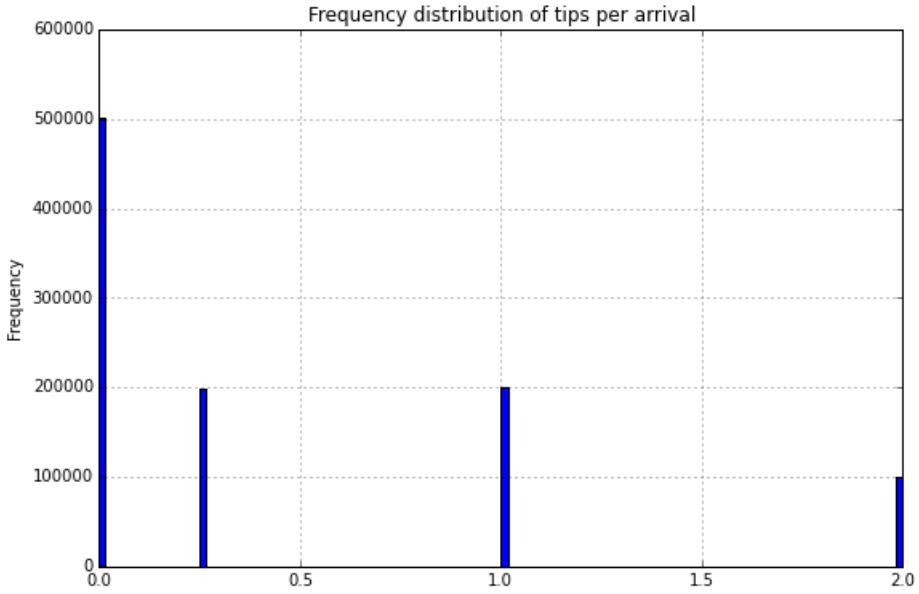

```
count    1000000.000000
mean         600.002708
std           30.007318
min          450.253214
25%          579.725321
50%          600.006192
75%          620.222092
max          739.678771
dtype: float64
```



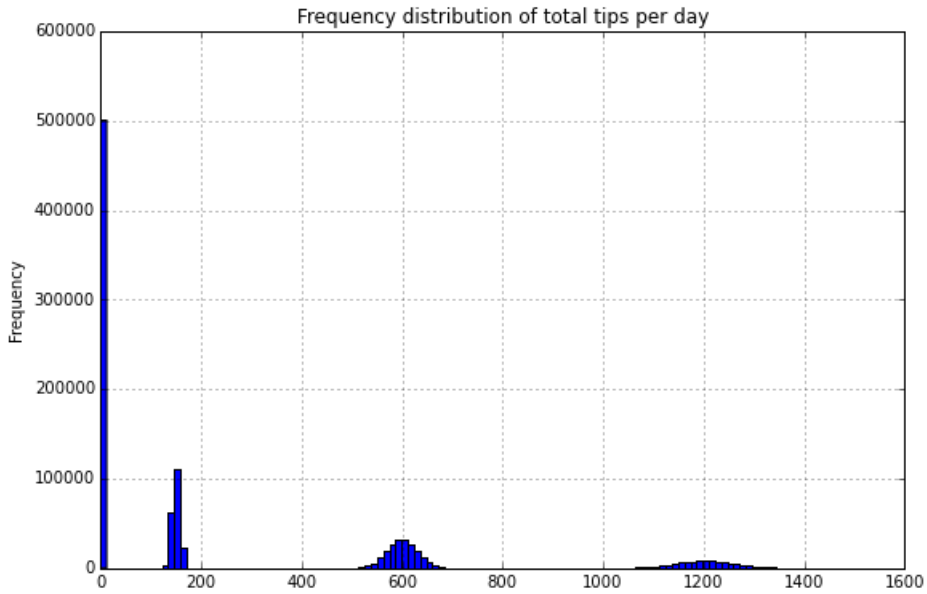

```
count    1000000.000000
mean           4.150777
std            0.593598
min            3.500000
25%            3.500000
50%            4.000000
75%            5.000000
max            5.000000
dtype: float64
```



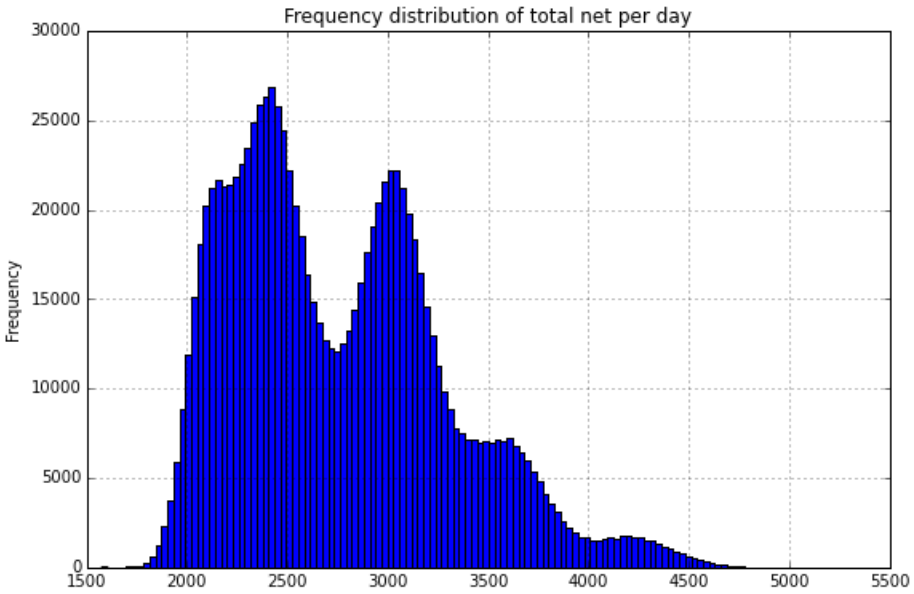

```
count    1000000.000000
mean        2490.487353
std          377.811927
min         1575.886249
25%         2183.533847
50%         2401.154755
75%         2855.214744
max         3691.195888
dtype: float64
```




```
count    1000000.000000
mean           0.450140
std            0.640479
min            0.000000
25%            0.000000
50%            0.000000
75%            1.000000
max            2.000000
dtype: float64
```




```
count    1000000.000000
mean         270.121213
std          385.088879
min            0.000000
25%            0.000000
50%            0.000000
75%          579.868929
max         1466.610766
dtype: float64
```




```
Out[6]:
count    1000000.000000
mean        2760.608566
std          542.424286
min         1575.886249
25%         2326.857391
50%         2668.203428
75%         3103.085025
max         5133.137682
dtype: float64
```

Examine each of the plots and the corresponding summary statistics. Note the following:

- The Normal distribution of the customer arrivals is as expected with a mean and median of 600.
- The distribution of profits per arrival appears as was observed previously.
- The distribution of total profits per day, is the vector product of arrivals per day and profits per arrival. Note, that this result is a complex distribution which would be difficult to handle except by simulation.
- The distribution of tips per arrival appears as was observed previously.
- The distribution of total tips per day is the vector product of arrivals per day and tips per arrival. Again, this result is a complex distribution which would be difficult to handle except by simulation.
- The distribution of the final total net profit per day is the sum of the distribution of total profits per day and the distribution of total tips per day. This final distribution is quite complex with five peaks.

6. Enter the following command to run the simulation again, this time assuming a mean of 1200 customers per day with a standard deviation of 20:

```
rs.sim_lemonade(100000, 1200, 20)
```

7. Review the resulting statistics and plots, comparing them with the simulation results for a 600 daily customer average.

**Note**: The distributions shown in this exercises are shown in terms of frequency of the values, not probability. The distribution would need to be normalized to transform frequency to probability values.

## Summary

This lab you have used either R or Python to simulate the income using a simple model for a lemonade stand. You have computed random draws from both named distributions and a special, or custom, distribution. You have computed the distributions of profits, tips and total net income. All of these distributions exhibit complex behavior which is only tractable using simulation.