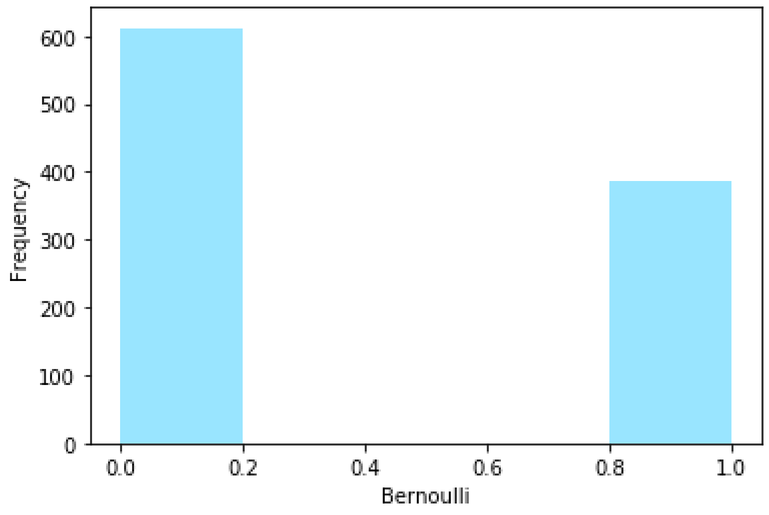
In [1]:
```python
#Bernoulli arrays with size 10, 100, 1000, and 1000000
from scipy.stats import bernoulli

p = 0.4

array1_bern = bernoulli.rvs(p, size=10)
array2_bern = bernoulli.rvs(p, size=100)
array3_bern = bernoulli.rvs(p, size=1000)
array4_bern = bernoulli.rvs(p, size=1000000)

#Calculates the mean of each arrays
array1_bern.mean()
array2_bern.mean()
array3_bern.mean()
array4_bern.mean()

#Calculates the standard deviation of the arrays
array1_bern.std()
array2_bern.std()
array3_bern.std()
array4_bern.std()
```

Out[1]:
```
0.490003399657594 26
```

The mean E(X)=p is the formula given in class, therefore the mean should be 0.4. The mean derived from the generated arrays is close to 0.4 but not exact. The bigger the size of the array the closer the mean to 0.4 is. This conclusion makes sense because the generator will not always simulate numbers for the p to be exactly 0.4 The standard deviation formula is the square root of p(1-p), therefore it should ideally be about 0.4899 From the generated arrays, it is close to 0.48

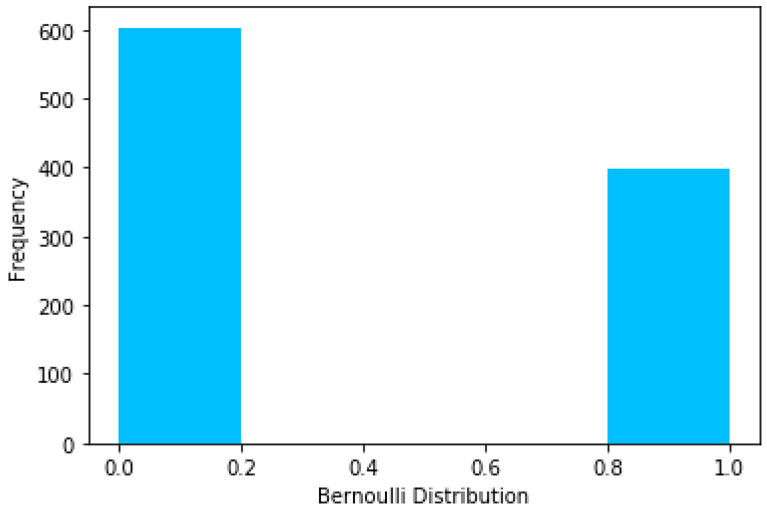
```
In [41]: # Histogram for Bernoulli Distribution
         import seaborn as sns

         data_bern1 = bernoulli.rvs(size=1000,p=0.4)
         ax= sns.distplot(data_bern1,
                          kde=False,
                          color="deepskyblue",)
         ax.set(xlabel='Bernoulli', ylabel='Frequency')
```

```
Out[41]: [Text(0, 0.5, 'Frequency'), Text(0.5, 0, 'Bernoulli')]
```



In [31]:
```
#PMF plot of Bernoulli Distribution
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns

p=0.4

data_bern2 = bernoulli.rvs(p, size=1000)
ax= sns.distplot(data_bern2,
                 kde=False,
                 color="deepskyblue",
                 hist_kws={"linewidth": 15,'alpha':1})
ax.set(xlabel='Bernoulli Distribution', ylabel='Frequency')
```

Out[31]:
```
[Text(0, 0.5, 'Frequency'), Text(0.5, 0, 'Bernoulli Distribution')]
```



Based on the PMF plot of the bernoulli distribution, we can see that the generated distribution is pretty close to how the distribution should ideally look like. We can visualize that there are only two possible outcomes, 0 and 1 or failure and success respectively. Since the probability parameter p is 0.4, ideally we should have 400 success and 600 failures which add up to our size 1000. In the generated histogram, we see that the successes are a little over 600 and the failures a little under 400, which both make sense.

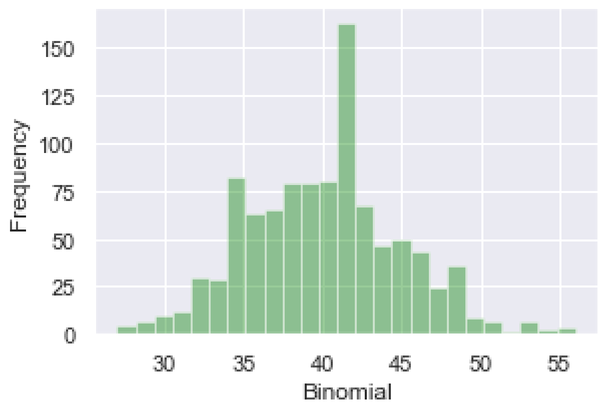
```
In [13]: #Binomial array with size 10, 100, 1000, and 1000000
         from scipy.stats import binom

         n = 100
         p = 0.4

         array1_binom = binom.rvs(n, p, size=10)
         array2_binom = binom.rvs(n, p, size=100)
         array3_binom = binom.rvs(n, p, size=1000)
         array4_binom = binom.rvs(n, p, size=1000000)

         #Calculates the mean of the arrays
         array1_binom.mean()
         array2_binom.mean()
         array3_binom.mean()
         array4_binom.mean()

         #Calculates the standard deviation of the arrays
         array1_binom.std()
         array2_binom.std()
         array3_binom.std()
         array4_binom.std()
```

```
Out[13]: 4.910387539931956
```

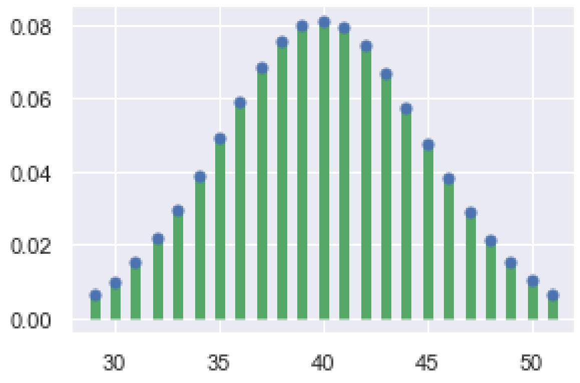
The mean E(X)=np which in this case equals to 40. In the generated arrays, the mean is also very close to 40. The standard deviation formula is the square root of np(1-p) which equals to approximately 4.89 This is also very close to the generated standard deviations.

```
In [172]: #Histogram for Binomial Distribution
          data_binom = binom.rvs(n=100,p=0.4,size=1000)
          ax = sns.distplot(data_binom,
                            kde=False,
                            color='green',)
          ax.set(xlabel='Binomial', ylabel='Frequency')
```

```
Out[172]: [Text(0, 0.5, 'Frequency'), Text(0.5, 0, 'Binomial')]
```



```
In [174]: #PMF plot of Binomial Distribution
          fig, ax = plt.subplots(1, 1)
          n = 100
          p = 0.4

          x = np.arange(binom.ppf(0.01, n, p),
                        binom.ppf(0.99, n, p))
          ax.plot(x, binom.pmf(x, n, p), 'bo', ms=5, label='binom pmf')
          ax.vlines(x, 0, binom.pmf(x, n, p), colors='g', lw=5)

Out[174]: <matplotlib.collections.LineCollection at 0x1a2192b1d0>
```

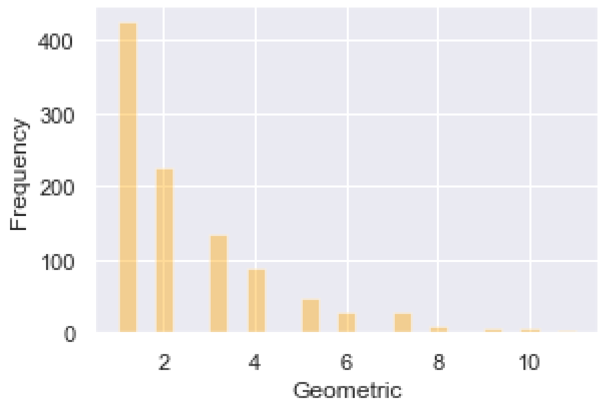


The binomial distribution is closely related to the bernoulli distribution because it has two outcomes, failure or success. However the binomial distribution has a number of repeated trials, in this case, it has n=100 trials. In our generated histogram, we notice that the 41 is abnormally higher than the other numbers. This is still normal because it means that out of 100 trials, 41 was more frequently generated. In the end, the generated distribution still has that bell curve visualization. The greater the n and the closer the p to 0.5, then the closer the distribution to a bell curve will look like.

```
In [14]: #Geometric array with size 10, 100, 1000, and 1000000
         from scipy.stats import geom
         import matplotlib.pyplot as plt

         p = 0.4

         array1_geom = geom.rvs(p, size=10)
         array2_geom = geom.rvs(p, size=100)
         array3_geom = geom.rvs(p, size=1000)
         array4_geom = geom.rvs(p, size=1000000)

         #Calculates the mean of the arrays
         array1_geom.mean()
         array2_geom.mean()
         array3_geom.mean()
         array4_geom.mean()

         #Calculates the standard deviation of the arrays
         array1_geom.std()
         array2_geom.std()
         array3_geom.std()
         array4_geom.std()

Out[14]: 1.93523847044475059
```

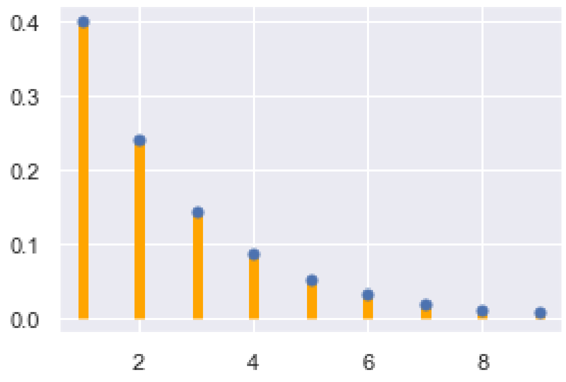
The mean of geometric distribution is 1/p which in this case equals to 2.5 The generated mean from the arrays gets closer and closer to 2.5 as the size increases. The standard deviation formula is the square root of (1-p)/(p^2) which equals to approximately 1.936 The generated standard deviations are close to that value.

```
In [164]: #Histogram for Geometric Distribution
          data_geom = geom.rvs(p=0.4, size=1000)
          ax = sns.distplot(data_geom,
                            kde=False,
                            color='orange')
          ax.set(xlabel='Geometric', ylabel='Frequency')

Out[164]: [Text(0, 0.5, 'Frequency'), Text(0.5, 0, 'Geometric')]
```



In [153]:
```
#PMF plot of Geometric Distribution
fig, ax = plt.subplots(1, 1)
p = 0.4

x = np.arange(geom.ppf(0.01, p),
              geom.ppf(0.99, p))
ax.plot(x, geom.pmf(x, p), 'bo', ms=5, label='geom pmf')
ax.vlines(x, 0, geom.pmf(x, p), colors='orange', lw=5)
```

Out[153]:
```
<matplotlib.collections.LineCollection at 0x1a20a62a20>
```



The geometric distribution represents the numbers of failures before you get the success. Since p=0.4, the probability of success is 0.4 and the probability of a failure is 1 - 0.4 = 0.6 which is the ideal case. In the histogram from the generated array, the success is a little above 400 which makes the success higher than ideally and the failures lower than ideally. Overall, the distribution looks very identical to the plot of PMF.

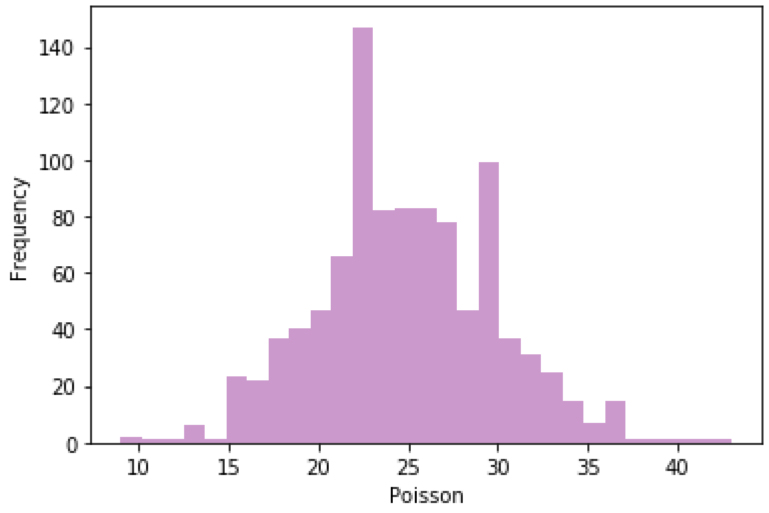
In [16]:
```
#Poisson array with size 10, 100, 1000, and 1000000
from scipy.stats import poisson

mu = 25

array1_poisson = poisson.rvs(mu, size=10)
array2_poisson = poisson.rvs(mu, size=100)
array3_poisson = poisson.rvs(mu, size=1000)
array4_poisson = poisson.rvs(mu, size=1000000)

#Calculates the mean of the arrays
array1_poisson.mean()
array2_poisson.mean()
array3_poisson.mean()
array4_poisson.mean()

#Calculates the standard deviation of the arrays
array1_poisson.std()
array2_poisson.std()
array3_poisson.std()
array4_poisson.std()
```

Out[16]:
```
4.998867905387279
```

The mean formula for poisson distribution is λT which in this case equals to 25. From the generated arrays, the mean is close to that value and as bigger the size the closer the value to the ideal mean. The standard deviation is the square root of the variance, which is the same as the mean. So the square root of 25 is 5. The standard deviation from the generated arrays is close to 5.

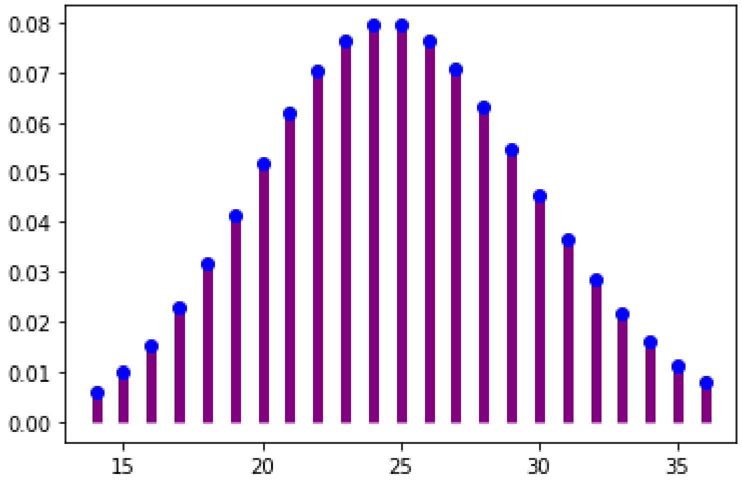
In [56]:
```
#Histogram for Poisson Distribution
import seaborn as sns

data_poisson = poisson.rvs(mu=25, size=1000)
ax = sns.distplot(data_poisson,
                  kde=False,
                  color='purple')
ax.set(xlabel='Poisson', ylabel='Frequency')
```

Out[56]:
```
[Text(0, 0.5, 'Frequency'), Text(0.5, 0, 'Poisson')]
```



In [49]:
```
#PMF plot of Poisson Distribution
import numpy as np

fig, ax = plt.subplots(1, 1)
mu = 25

x = np.arange(poisson.ppf(0.01, mu),
              poisson.ppf(0.99, mu))
ax.plot(x, poisson.pmf(x, mu), 'bo', ms=6, label='poisson pmf')
ax.vlines(x, 0, poisson.pmf(x, mu), colors='purple', lw=5)
```

Out[49]:
```
<matplotlib.collections.LineCollection at 0x1a293d2978>
```



The poisson distribution models the number of events occurring within a given time interval. It is derived from the binomial distribution, hence it has a similar shape. In the generated histogram from the array, the shape is similar to the PMF except not exact since it is much higher in the 23's even though the middle, 25 or the mu should be the highest point ideally. However, the generated histogram cannot always look exactly like the PMF.

```
In [3]: #Application - Customer traffic numbers for an entire year
        mu = 200
        application = poisson.rvs(mu, size=365)
        application
```

```
Out[3]: array([202, 232, 200, 215, 218, 214, 171, 187, 208, 176, 205, 191, 195,
               202, 199, 244, 172, 190, 172, 189, 183, 192, 181, 194, 170, 191,
               199, 190, 186, 183, 197, 195, 186, 216, 194, 202, 207, 187, 200,
               204, 196, 188, 193, 204, 201, 218, 207, 180, 219, 183, 192, 167,
               190, 214, 218, 196, 223, 208, 202, 180, 200, 194, 208, 208, 203,
               213, 176, 197, 179, 211, 216, 202, 208, 195, 194, 188, 212, 205,
               180, 207, 212, 206, 199, 210, 209, 222, 217, 182, 209, 204, 202,
               199, 206, 213, 227, 207, 197, 191, 218, 190, 194, 180, 171, 189,
               210, 200, 213, 185, 205, 181, 200, 202, 218, 179, 207, 209, 178,
               202, 202, 196, 222, 195, 201, 214, 196, 234, 220, 212, 200, 189,
               195, 181, 220, 204, 226, 183, 226, 188, 195, 211, 217, 188, 212,
               180, 219, 187, 216, 214, 194, 197, 181, 190, 215, 215, 208, 205,
               182, 184, 208, 196, 187, 190, 192, 208, 182, 199, 193, 186, 190,
               182, 195, 203, 186, 195, 176, 211, 213, 208, 203, 194, 180, 203,
               183, 193, 191, 198, 206, 201, 216, 201, 216, 174, 190, 198, 193,
               197, 213, 191, 218, 182, 191, 204, 181, 209, 182, 203, 208, 177,
               201, 197, 194, 219, 201, 204, 187, 199, 220, 207, 210, 213, 215,
               237, 206, 196, 195, 187, 203, 190, 189, 215, 187, 211, 184, 198,
               201, 202, 209, 222, 200, 181, 214, 217, 216, 171, 198, 215, 211,
               197, 205, 196, 208, 197, 190, 198, 212, 228, 196, 203, 198, 194,
               204, 211, 189, 176, 194, 198, 220, 201, 204, 214, 200, 189, 219,
               186, 185, 208, 192, 169, 191, 186, 193, 187, 232, 186, 181, 185,
               185, 216, 204, 202, 192, 183, 200, 213, 222, 211, 209, 194, 206,
               186, 174, 195, 216, 194, 171, 196, 202, 203, 213, 194, 213, 217,
               197, 226, 191, 179, 179, 184, 227, 210, 211, 193, 202, 210, 200,
               196, 223, 186, 177, 203, 175, 228, 206, 242, 199, 208, 183, 205,
               203, 209, 194, 212, 221, 182, 212, 191, 189, 201, 220, 207, 239,
               181, 227, 176, 207, 213, 186, 221, 180, 210, 174, 201, 209, 195,
               194])
```

```
In [10]: #Calculates how much money you should expect to make daily on average fr
         om distribution
         sales = application*29.95
         sum_sales = sales.sum()
         sum_sales
         average_per_year = sum_sales/365
         average_per_year
```

```
Out[10]: 5983.107397260274
```

Since this is a poisson ditribution the expected value should equal to the number of customers daily times the price of the candle. So the ideal number of customers daily is 200 and the price is 29.95 which means the expected value should be (200)(29.95)= 5990. This value is similar and close to the generated average daily revenue. It is not exact because in the simulation, we did not always have an exact number of 200 customers every day in a whole year, which is a realistic way to look at this problem.

```
In [44]: #Calculates the 95th percentile of the distribution
         np.percentile(application,95)
```

```
Out[44]: 222.0
```