STOR 320 Intro to Data Science Midterm Exam

Please submit the solution to gradescope by 6:00 AM, Oct 10, Thursday.

Name: Ivy Nangalia

PID: 730670491

In this exam, you will explore relations between exponential distribution and Poisson process using visualization methods.

For each problem, you are required to **display** your results (**tables, matrix, plots, or variables**) in the Jupyter notebook. **(100 points)**

Instructors will not answer any questions regarding the midterm exam during the exam period. Please answer the questions based on your understanding of the problems.

1. Create a variable callend `lambda_param` and assign 1 to this variable. (1 point)

2. Generate one random variable that follows the exponential distribution with rate `lambda_param`. (1 point)

Hint: You can use `np.random.exponential(scale = 1 / lambda)` to generate an exponential random variable with rate `lambda` and mean `1 / lambda`.

3. Generate a 10,000 * 1,000 numpy matrix, where each entry follows the exponential distribution with rate `lambda_param`. (5 points)

4. Convert the above numpy matrix to a DataFrame named `df_exp` with 10,000 rows and 1,000 columns.

Rename the column names of DataFrame to `Trial_1, Trial_2, Trial_3, ..., Trial_1000`. (5 points)

In [ ]:
```
#1
lambda_param = 1
lambda_param
```

Out[ ]:
```
1
```

In [ ]:
```
#2
import numpy as np
np.random.seed(3) # to ensure analysis is correct with export
```

In [ ]:
```
x = np.random.exponential(scale = 1 / lambda_param)
x
```

Out[ ]:
```
0.8002823868824798
```

In [ ]:
```
#3
rand_matrix = np.random.exponential(scale=1/lambda_param, size=(10000, 1000))
rand_matrix
```

Out[ ]:
```
array([[1.23150785, 0.3437654 , 0.71504031, ..., 0.03579858, 0.13318147,
        0.33015213],
       [0.40860371, 0.38207235, 0.8335838 , ..., 0.5574183 , 1.60808522,
        2.70759774],
       [1.61806613, 1.00197754, 0.83415624, ..., 0.2182116 , 0.21483518,
        0.12269318],
       ...,
       [2.18676831, 1.94108969, 0.78127237, ..., 0.23454046, 0.40772307,
        1.26286542],
       [0.95397083, 1.21062992, 0.6094575 , ..., 1.51145186, 0.17298346,
        0.78546511],
       [1.13293674, 2.22612195, 2.14500175, ..., 0.42271515, 0.42854001,
        1.09411358]])
```

In [ ]:
```
#4
import pandas as pd
df_exp = pd.DataFrame(rand_matrix)

#creating list of column names (1-1000)
```

```
colnames = []
for i in range(1,1001):
    colnames.append(f"Trial_{i}")

df_exp.columns = colnames
df_exp
```

Out[ ]:

	Trial_1	Trial_2	Trial_3	Trial_4	Trial_5	Trial_6	Trial_7	Trial_8	Trial_9	Trial_10	...	Trial_991	Trial_992	Trial_9
0	1.231508	0.343765	0.715040	2.234431	2.266187	0.134201	0.232238	0.052839	0.581266	0.030332	...	1.490284	3.576348	0.2718
1	0.408604	0.382072	0.833584	1.065470	1.863135	0.477872	2.780541	0.479758	0.550512	0.534882	...	1.241636	1.476101	1.7379
2	1.618066	1.001978	0.834156	0.195109	1.283215	0.368328	1.069578	0.781567	0.670035	1.256192	...	1.162576	0.925869	1.5962
3	2.246240	1.450221	1.228864	2.850949	0.132274	1.884414	0.529841	0.615969	0.149409	0.406539	...	0.655275	2.446376	0.282
4	0.614536	1.095680	2.240833	2.734598	0.426754	0.127801	0.646907	0.598755	2.493809	2.533941	...	0.071570	0.233765	0.450
...	...	...	...	...	...	...	...	...	...	...	...	...	...	
9995	0.286150	0.718906	0.149540	0.414370	0.492995	1.387779	2.706047	0.263052	0.118169	0.572166	...	0.037271	0.411919	0.8959
9996	4.782403	0.319942	0.197791	0.814149	0.520699	0.441888	0.814120	0.003616	2.657248	0.378557	...	0.995515	0.094233	2.5358
9997	2.186768	1.941090	0.781272	1.404360	0.229703	0.851232	0.371126	0.153869	0.946485	0.135969	...	0.962214	1.058431	0.8009
9998	0.953971	1.210630	0.609457	1.301056	0.942157	0.094473	0.009445	1.289949	2.582395	0.355277	...	0.314475	0.004188	1.320
9999	1.132937	2.226122	2.145002	0.012613	0.369454	0.701227	0.645073	0.772488	0.148551	0.930125	...	1.838705	0.838885	0.2478

10000 rows × 1000 columns

In [ ]:

```
#creating list of column names (1-1000)
colnames = []
for i in range(1,1001):
    colnames.append(f"Trial_{i}")

df_exp.columns = colnames
df_exp.head()
```

Out[ ]:

	Trial_1	Trial_2	Trial_3	Trial_4	Trial_5	Trial_6	Trial_7	Trial_8	Trial_9	Trial_10	...	Trial_991	Trial_992	Trial_993
0	0.082540	0.022694	0.227935	0.003309	0.022686	0.001189	0.142899	0.152457	0.014423	0.003001	...	0.337542	0.002815	0.291912
1	0.046929	0.044728	0.552958	0.048669	0.068577	0.054643	0.269894	0.115137	0.096899	0.076375	...	0.330064	0.094434	0.001087
2	0.153466	0.082353	0.102819	0.023043	0.104888	0.049535	0.436083	0.307848	0.336352	0.103161	...	0.049360	0.391706	0.196125
3	0.134225	0.021248	0.004459	0.210301	0.005124	0.179952	0.487525	0.074705	0.005490	0.021389	...	0.128213	0.015710	0.031664
4	0.454298	0.399104	0.102862	0.005322	0.041708	0.025114	0.229330	0.128549	0.434298	0.180451	...	0.257307	0.036509	0.261321

5 rows × 1000 columns

5. Create a new DataFrame named `df_cumsum` with the same dimensions as `df_exp`. (10 points)

For each column in `df_exp`, compute the cumulative sum and store it in the corresponding column of `df_cumsum`.

Let the row index start from 1, instead of 0.

Hint: Use function `cumsum()`.

In [ ]:

```
#5
df_cumsum = df_exp.cumsum()
df_cumsum = df_cumsum.set_index(pd.RangeIndex(1,len(df_cumsum)+1))
df_cumsum.head()
```

Out[ ]:

	Trial_1	Trial_2	Trial_3	Trial_4	Trial_5	Trial_6	Trial_7	Trial_8	Trial_9	Trial_10	...	Trial_991	Trial_992	Trial_993
1	1.231508	0.343765	0.715040	2.234431	2.266187	0.134201	0.232238	0.052839	0.581266	0.030332	...	1.490284	3.576348	0.271899
2	1.640112	0.725838	1.548624	3.299900	4.129321	0.612072	3.012779	0.532597	1.131777	0.565214	...	2.731920	5.052449	2.009866
3	3.258178	1.727815	2.382780	3.495010	5.412537	0.980400	4.082357	1.314164	1.801812	1.821406	...	3.894496	5.978318	3.606106
4	5.504417	3.178036	3.611644	6.345959	5.544811	2.864814	4.612198	1.930132	1.951221	2.227945	...	4.549771	8.424694	3.888177
5	6.118953	4.273717	5.852477	9.080556	5.971565	2.992615	5.259105	2.528888	4.445029	4.761886	...	4.621341	8.658458	4.338920

5 rows × 1000 columns

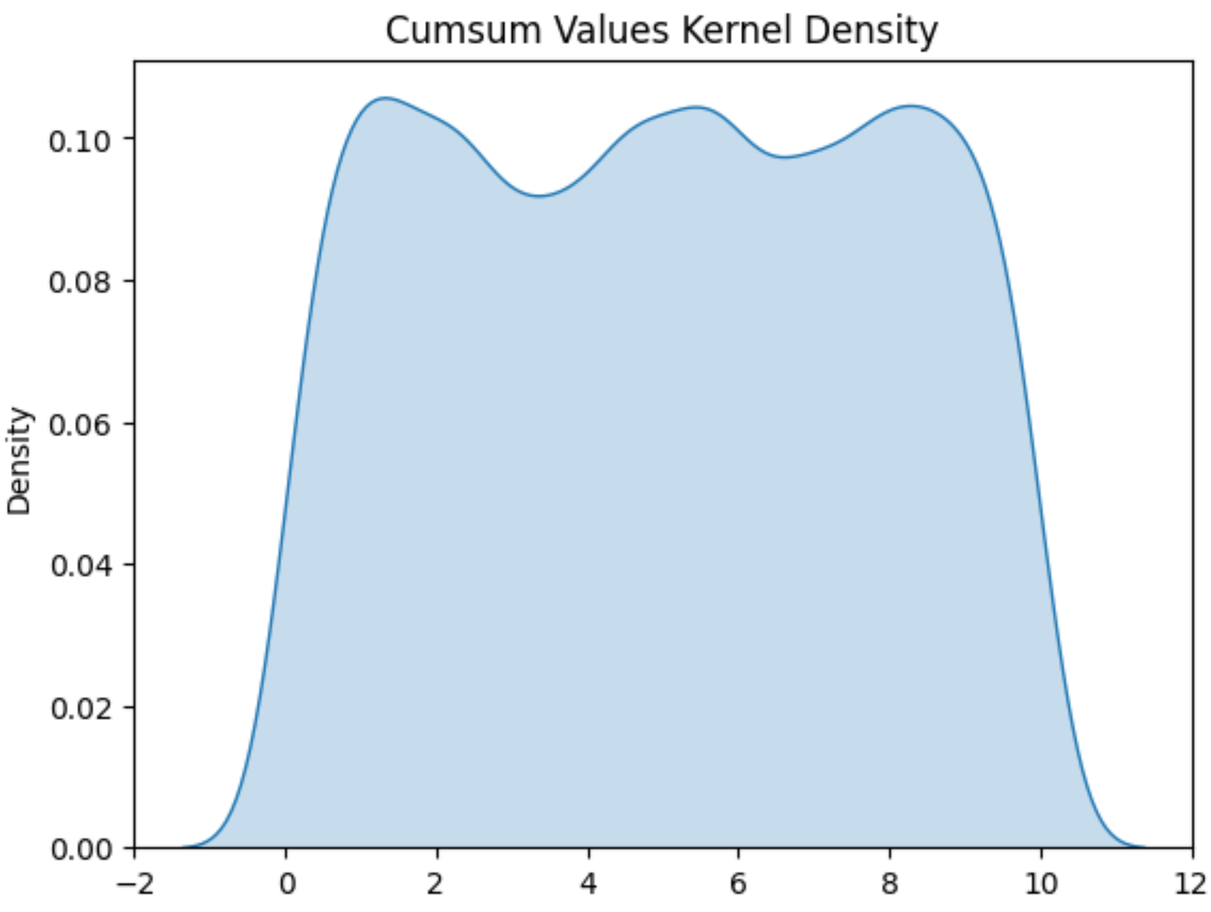
6 Compare with uniform distribution.

- Extract all values from `df_cumsum` that are less than 10 and flatten them into a one-dimensional array named `cumsum_values_lt_10`. (5 points)
- Plot the density distribution of `cumsum_values_lt_10` using `sns.histplot`. (5 points)
- Generate a numpy array named `uniform_values` consisting of 100,000 variables that follow the uniform distribution between [0, 10]. On the same figure, plot the density distribution of `uniform_values`. (10 points)
- In a brief paragraph, describe any similarities or differences you observe between these two distributions. (4 points)

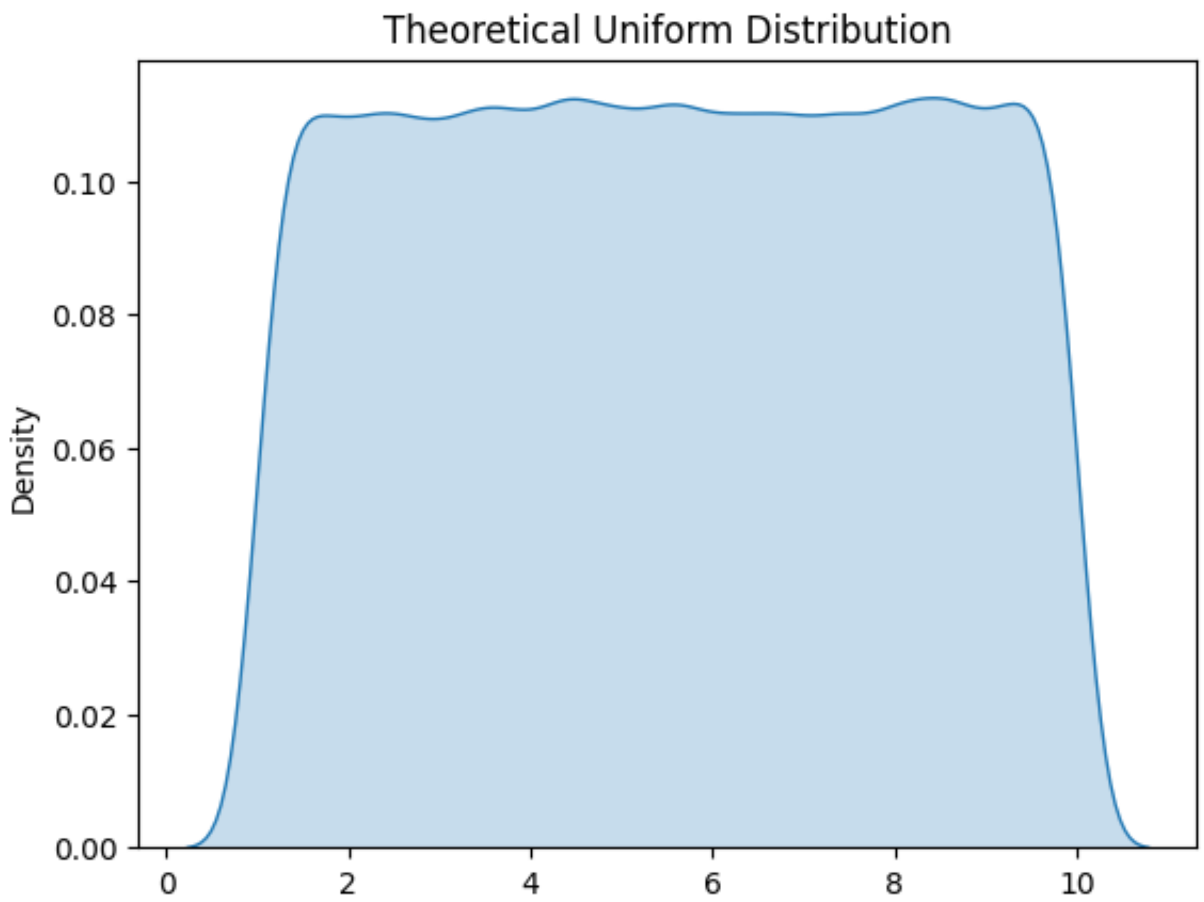
Hint: You can use `np.random.uniform` to generate uniform distribution.

In [ ]:
```
import seaborn as sns
import matplotlib.pyplot as plt
```

In [ ]:
```
cumsum_values_lt_10 = df_cumsum[df_cumsum < 10].values.flatten()
sns.kdeplot(data=cumsum_values_lt_10, fill=True)
plt.title("Cumsum Values Kernel Density");
```



In [ ]:
```
uniform_values = np.random.uniform(low=1.0, high=10.0, size=100000)
sns.kdeplot(data=uniform_values, fill=True)
plt.title("Theoretical Uniform Distribution");
```



The two plots have the same general plateau-esque shape on the sides, however, the cumsum values do not appear to follow the same uniform distribution, as there are many peaks and valleys in a way that is not observed in the uniform graph: the cumsum graph appears to be trimodal, whereas the theoretical uniform distribution is unimodal. The range of the Cumsum values appears to be higher, going from -2 to 12, whereas the uniform values only go from 1 to 10.

7. Create `max_indices` (10 points)

- For each column in `df_cumsum`, find the maximum row index where the cumulative sum is less than or equal to 10.
- Create a Series named `max_indices` containing these indices for all 1,000 columns.

In [ ]:
```
max_indicies = df_cumsum[df_cumsum <= 10].idxmax(axis=0)
max_indicies
```

Out[ ]:
```
Trial_1        8
Trial_2       10
Trial_3       13
Trial_4        5
Trial_5       14
              ..
Trial_996     15
Trial_997      7
Trial_998     13
Trial_999      7
Trial_1000     8
Length: 1000, dtype: int64
```

8. Compare with Poisson distribution

- Generate 100,000 Poisson-distributed random variables using `numpy.random.poisson` with parameter `λ =10 ∗ lambda_param`. Store the result in `poisson_samples`. (5 points)
- Plot the density distribution of the `max_indices` obtained in Problem 7 using `sns.histplot`. (5 points)
- On the same figure, plot the distribution of `poisson_samples`. (10 points)
- Compare the empirical distribution of max_indices with the theoretical Poisson distribution. Provide observations about the comparison in one paragraph. (5 points)

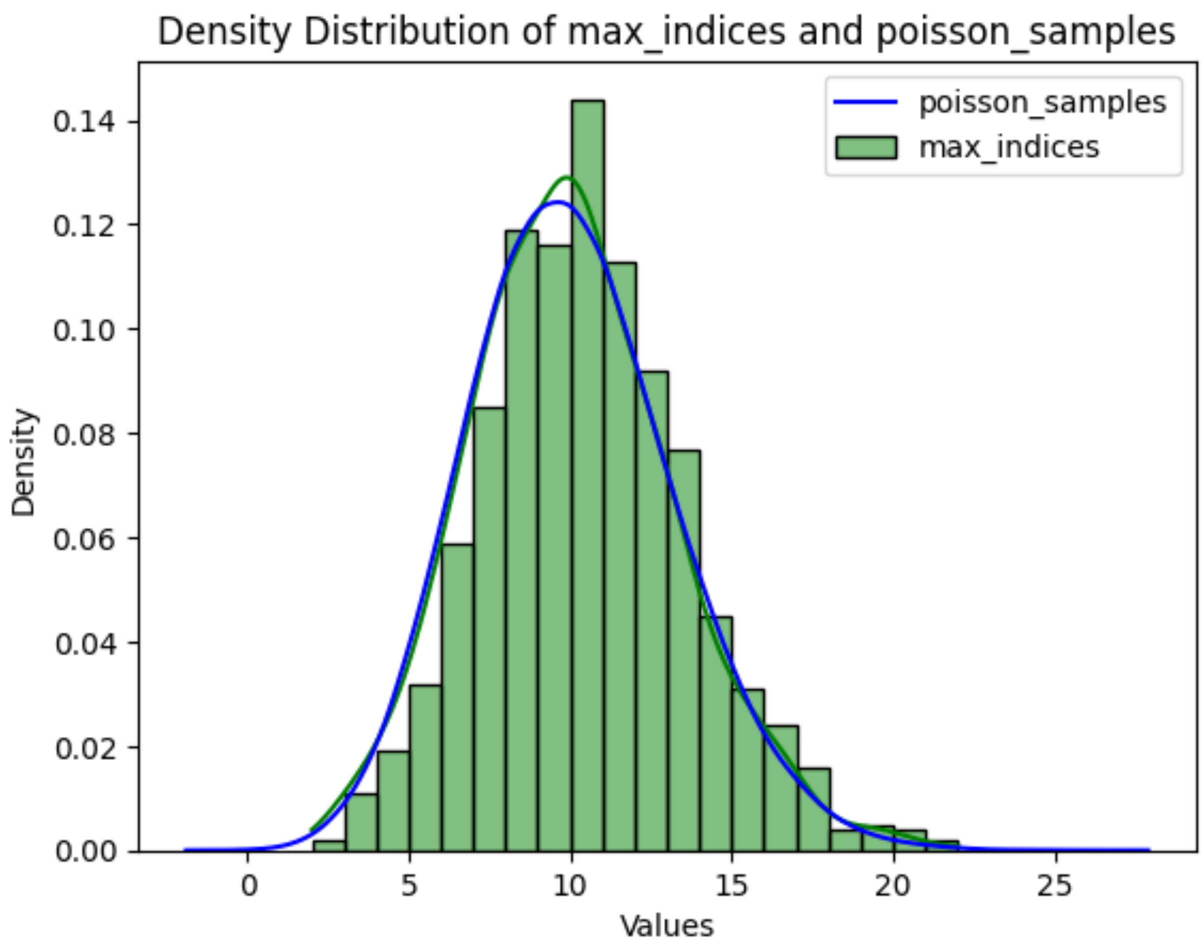
Hint: You can use `poisson_samples = np.random.poisson(lam=lambda_poisson, size=sample_size)` to generate Poisson distribution.

In [ ]:
```
import matplotlib.pyplot as plt
import seaborn as sns
```

In [ ]:
```
poisson_samples = np.random.poisson(lam=(10 * lambda_param), size=100000)

sns.histplot(data=max_indicies, stat="density", kde=True, color="green", label="max_indices", bins=20)
sns.kdeplot(data=poisson_samples, color="blue", label="poisson_samples",
            bw_adjust=2) # to fix the shape of the theoretical curve
plt.title('Density Distribution of max_indices and poisson_samples')
```

```
plt.xlabel('Values')
plt.ylabel('Density')
plt.legend()
plt.subplot()
plt.show()
```



The KDE of the empirical `max_indecies` data closely matches that of the theoretical poisson distribution. The data does appear to follow the same pattern. However, the peak of the empirical data is higher than expected, and extreme values on the right side of the mean were observed more commonly. There also appears to be a left skew. Nonetheless, for empirical data, the data follow the patterm quite strongly, and it is likely that they come from a poisson distribution.

9. In the DataFrame `df_exp`:

- Calculate the average of each column for the first 10 rows. How many columns have an average value that is between [0.9,1.1]? (1 point)
- Calculate the average of each column for the first 100 rows. How many columns have an average value that is between [0.9,1.1]? (1 point)
- Calculate the average of each column for the first 1000 rows. How many columns have an average value that is between [0.9,1.1]? (1 point)
- What do you observe from the above calculation? (1 point)

In [ ]:
```
def test(x: float) -> int:
    """Seeing if a number is between [0.9, 1.1]"""
    if 0.9 <= x <= 1.1:
        return 1
    else:
        return 0
```

In [ ]:
```
first_10_avg = df_exp[0:10].mean(axis=1)
count = 0
for i in first_10_avg:
    count += test(i)
count
```

Out[ ]: 10

In [ ]:
```
first_100_avg = df_exp[0:100].mean(axis=1)
count = 0
for i in first_100_avg:
    count += test(i)
count
```

Out[ ]: 100

In [ ]:
```
first_1000_avg = df_exp[0:1000].mean(axis=1)
count = 0
for i in first_1000_avg:
    count += test(i)
count
```

Out[ ]: 999

In [ ]:
```
# testing entire distribution to confirm the pattern
df_avg = df_exp.mean(axis=1)
count = 0
for i in df_avg:
    count += test(i)
print(f"{count}/{len(df_avg)}")
```

```
9984/10000
```

I observe that the vast majority (~99%) of the columns has an average value between 0.9 and 1.1, and I'm sure that as the number of observations approaches infinity, the true percentage outside of that range will be reached.

If the data follows a normal distribution, and the mean is 1 and 99.7% of the observations are within 0.1 of the mean, the standard deviation is approximately $\frac{1}{300}$ according to the empirical rule, however it is not proven that any of these conditions are met.

10. Repetition with `lambda_rate = 5`. (10 points)

Repeat Problems **1 to 8**, but this time use a rate parameter `lambda_rate = 5` instead of `lambda_rate = 1`.

Provide the code and two new distribution plots for the cumulative value (with uniform distribution) and maximum row index (with Poisson distribution).

In [ ]:
```
lambda_rate = 5

# 1 asked us to define lambda_param
# so i'm setting lambda_rate = lambda_param with the new value

lambda_param = lambda_rate
lambda_param
```

Out[ ]: 5

In [ ]:
```
#10.2
x = np.random.exponential(scale = 1 / lambda_param)
x
```

Out[ ]: 0.25171373662762503

In [ ]:
```
#10.3
rand_matrix = np.random.exponential(scale=1/lambda_param, size=(10000, 1000))
rand_matrix
```

Out[ ]:
```
array([[0.0825401 , 0.02269397, 0.22793497, ..., 0.31100629, 0.13531948,
        0.13188323],
       [0.04692942, 0.04472832, 0.55295841, ..., 0.13761075, 0.02457702,
        0.38212077],
       [0.15346603, 0.08235288, 0.10281897, ..., 0.1533629 , 0.52669495,
        0.19910331],
       ...,
       [0.093147  , 0.08718414, 0.20408623, ..., 0.06529774, 0.15596975,
        0.50815503],
       [0.60196186, 0.05134152, 0.03722916, ..., 0.44398238, 0.15148455,
        0.0073037 ],
       [0.03989201, 0.03012943, 0.27479173, ..., 0.01190892, 0.38543737,
        0.16149646]])
```

In [ ]:
```
#10.4
import pandas as pd
df_exp = pd.DataFrame(rand_matrix)

#creating list of column names (1-1000)
colnames = []
for i in range(1,1001):
    colnames.append(f"Trial_{i}")

df_exp.columns = colnames
df_exp.head(5)
```

Out[ ]:

	Trial_1	Trial_2	Trial_3	Trial_4	Trial_5	Trial_6	Trial_7	Trial_8	Trial_9	Trial_10	...	Trial_991	Trial_992	Trial_993
0	0.082540	0.022694	0.227935	0.003309	0.022686	0.001189	0.142899	0.152457	0.014423	0.003001	...	0.337542	0.002815	0.291912
1	0.046929	0.044728	0.552958	0.048669	0.068577	0.054643	0.269894	0.115137	0.096899	0.076375	...	0.330064	0.094434	0.001087
2	0.153466	0.082353	0.102819	0.023043	0.104888	0.049535	0.436083	0.307848	0.336352	0.103161	...	0.049360	0.391706	0.196125
3	0.134225	0.021248	0.004459	0.210301	0.005124	0.179952	0.487525	0.074705	0.005490	0.021389	...	0.128213	0.015710	0.031664
4	0.454298	0.399104	0.102862	0.005322	0.041708	0.025114	0.229330	0.128549	0.434298	0.180451	...	0.257307	0.036509	0.261321

5 rows × 1000 columns

In [ ]:

```
#10.5
df_cumsum = df_exp.cumsum()
df_cumsum = df_cumsum.set_index(pd.RangeIndex(1,len(df_cumsum)+1))
df_cumsum.head()
```

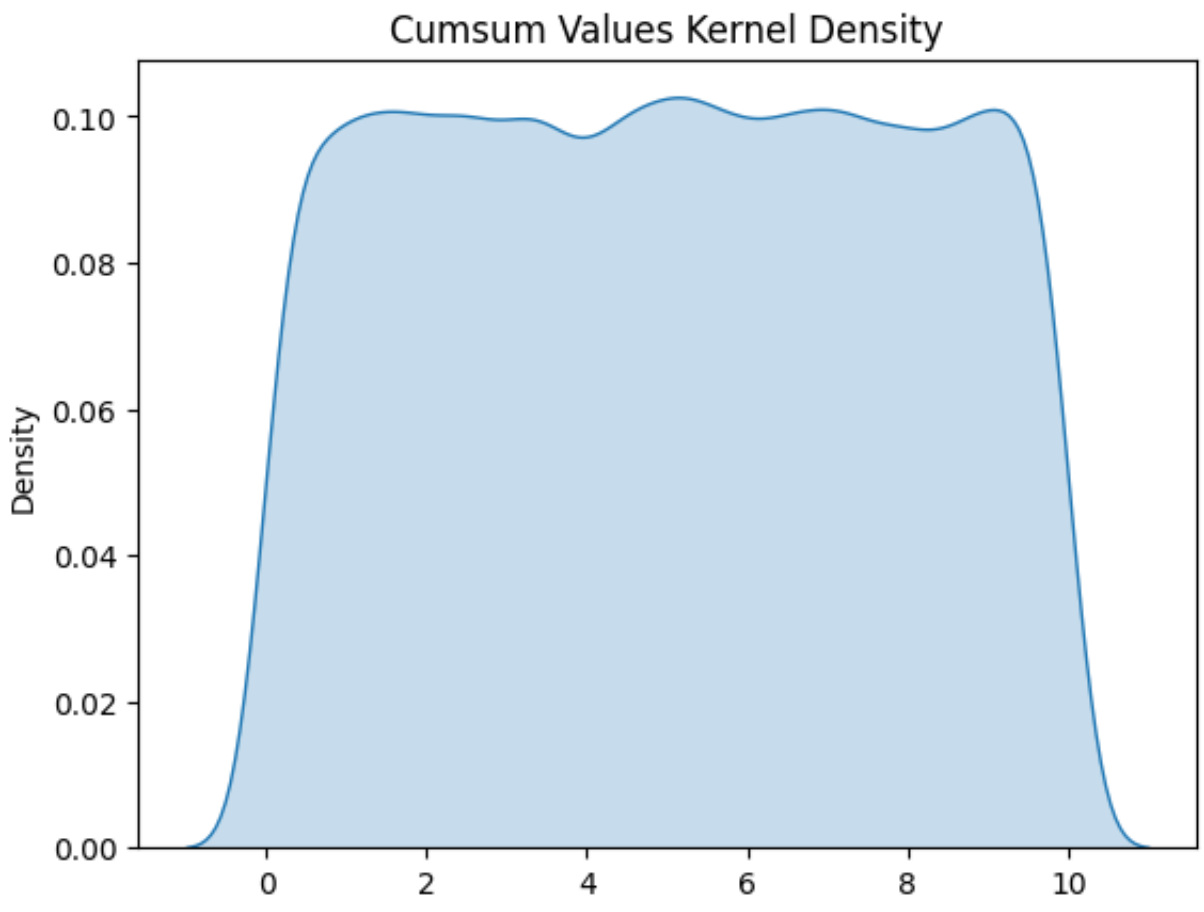
Out[ ]:

	Trial_1	Trial_2	Trial_3	Trial_4	Trial_5	Trial_6	Trial_7	Trial_8	Trial_9	Trial_10	...	Trial_991	Trial_992	Trial_993
1	0.082540	0.022694	0.227935	0.003309	0.022686	0.001189	0.142899	0.152457	0.014423	0.003001	...	0.337542	0.002815	0.291912
2	0.129470	0.067422	0.780893	0.051978	0.091263	0.055832	0.412793	0.267594	0.111322	0.079377	...	0.667606	0.097249	0.292999
3	0.282936	0.149775	0.883712	0.075021	0.196151	0.105367	0.848876	0.575442	0.447674	0.182537	...	0.716966	0.488955	0.489123
4	0.417160	0.171023	0.888171	0.285321	0.201275	0.285320	1.336400	0.650147	0.453164	0.203927	...	0.845179	0.504666	0.520788
5	0.871458	0.570127	0.991033	0.290644	0.242982	0.310434	1.565731	0.778697	0.887462	0.384378	...	1.102486	0.541175	0.782109

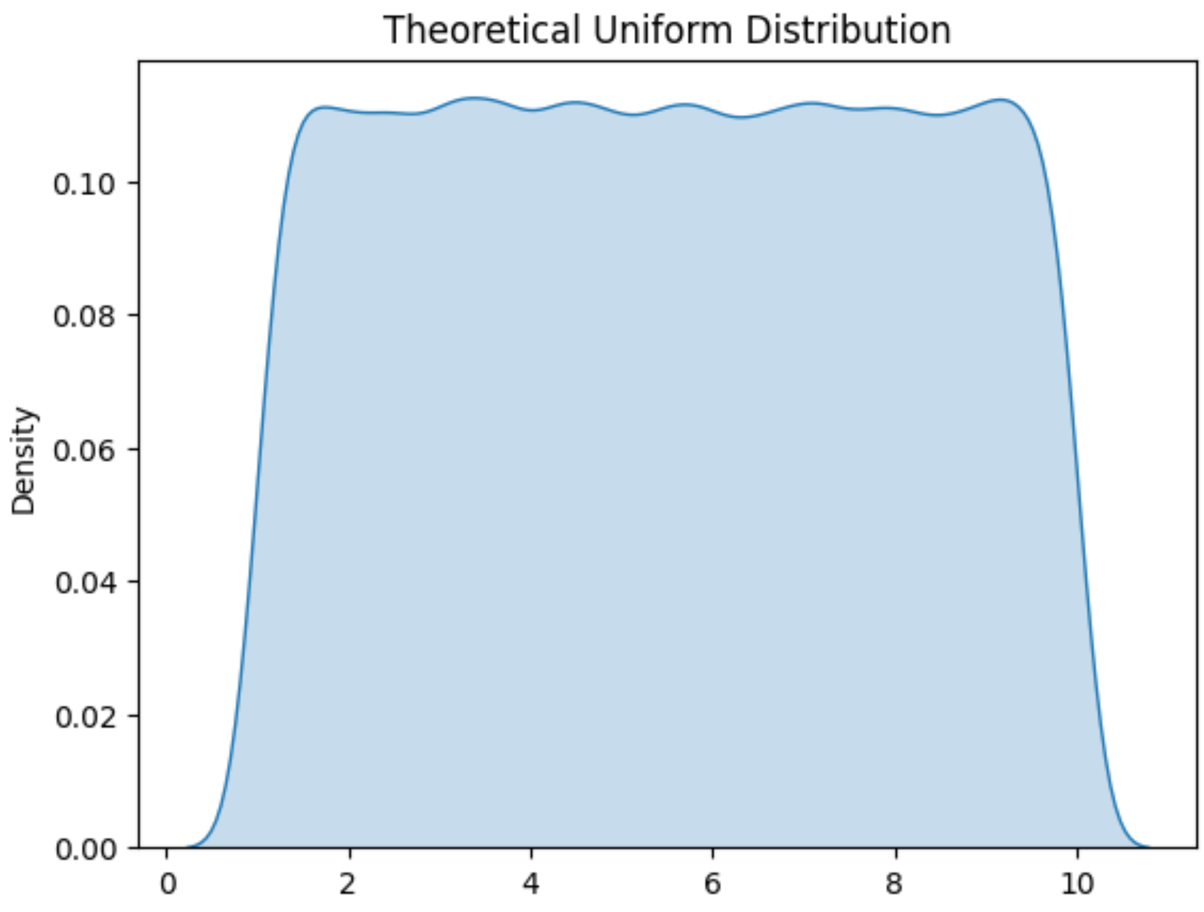
5 rows × 1000 columns

In [ ]:

```
#10.6
cumsum_values_lt_10 = df_cumsum[df_cumsum < 10].values.flatten()
sns.kdeplot(data=cumsum_values_lt_10, fill=True)
plt.title("Cumsum Values Kernel Density");
```



In [ ]:
```
#10.6
uniform_values = np.random.uniform(low=1.0, high=10.0, size=100000)
sns.kdeplot(data=uniform_values, fill=True)
plt.title("Theoretical Uniform Distribution");
```



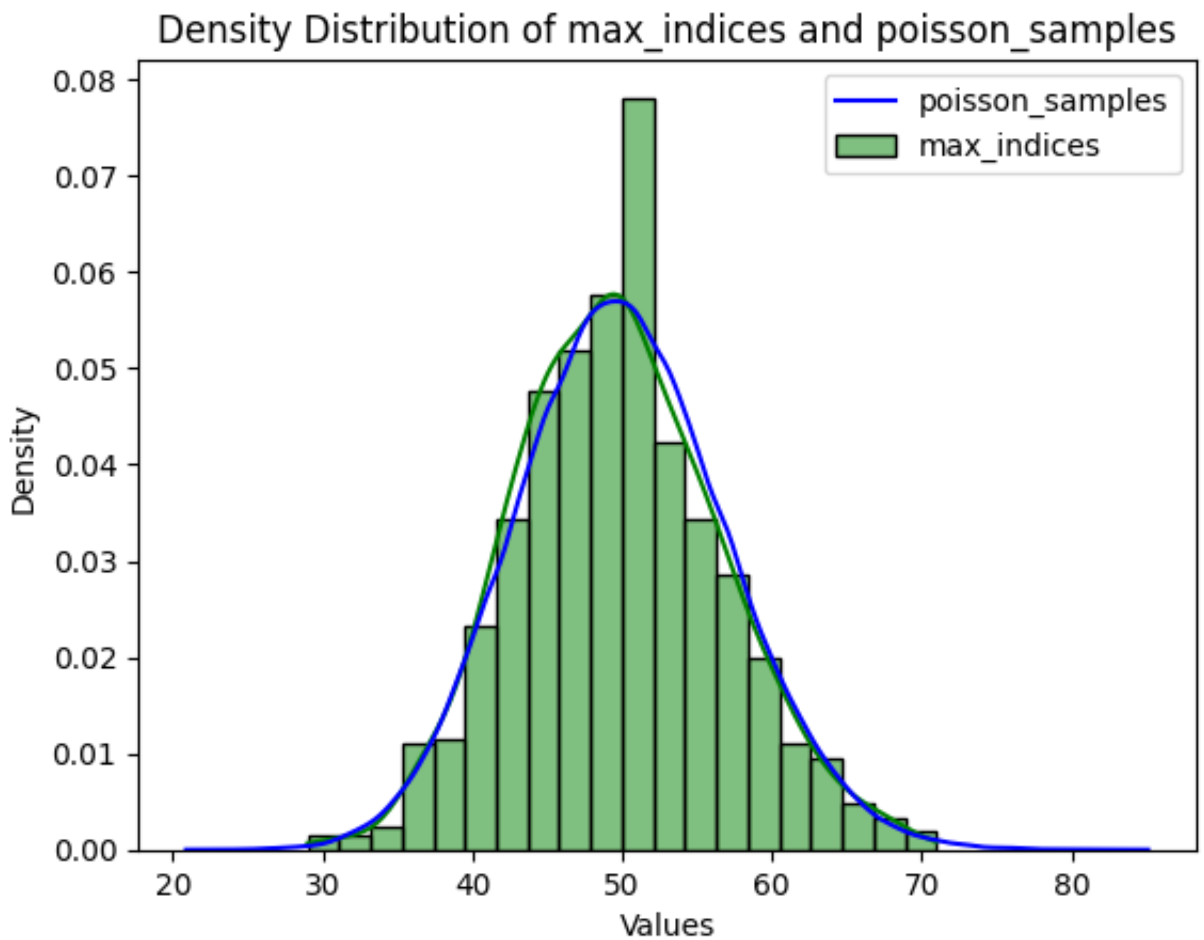
The distribution with the new lambda value appears to follow the uniform distribution better. It has the same range as the uniform distribution, and the same behavior on the sides. There does not appear to be the same extreme peaks and valleys, although the curve is not as flat as the theoretical uniform distribution.

In [ ]:
```
#10.7
max_indicies = df_cumsum[df_cumsum <= 10].idxmax(axis=0)
max_indicies
```

Out[ ]:
```
Trial_1       60
Trial_2       62
Trial_3       60
Trial_4       56
Trial_5       51
              ..
Trial_996     61
Trial_997     46
Trial_998     44
Trial_999     44
Trial_1000    40
Length: 1000, dtype: int64
```

In [ ]:
```
#10.8
poisson_samples = np.random.poisson(lam=(10 * lambda_param), size=100000)

sns.histplot(data=max_indicies, stat="density", kde=True, color="green", label="max_indices", bins = 20)
sns.kdeplot(data=poisson_samples, color="blue", label="poisson_samples")
plt.title('Density Distribution of max_indices and poisson_samples')
plt.xlabel('Values')
plt.ylabel('Density')
plt.legend()
plt.subplot()
plt.show()
```



There appears to be a right skew in the data, and a large peak at around 50. Besides these anomalies, the max-indicies appear to follow a poisson distribution, and the KDE lines mostly match up aside from the previously mentioned skew. The max_indecies appear to be unimodal and have a smaller range than the poisson theoretical distribution.

11. Application to Hospital Operations

In hospital operations models, we usually assume the number of patient arrivals in an hour follows the Poisson distribution.

- In the setting of hospital, suppose time intervals between two consecutive patients follow the exponential distribution. Then, what is the meaning of the max row index we obtained in `max_indices` in the setting of hospital operations? (3 points)

- Based on your observations from the previous problems, discuss in what settings it is reasonable to assume that the distribution of new patients arriving at a hospital follows a Poisson distribution. (2 points)

If each row index represents the time interval between the nth and the n+1th patient, then the max row index shows the patients with the largest interval between them, with the value at that index indicating the time between those two.

It would be reasonable to assume that the distribution of patient visits follows the Poisson distribution if they meet the criteria for that distribution. These criteria (in context) are:

- Independence: Each patient enters independently of one another.
- Constant probability: The probability of each patient entering does not change over time.
- There is no limit to the number of times that a patient can enter.
- Proportionality: It would be twice as likely for a patient to walk in over the course of two hours than one.

It is unlikely that a hospital can meet all of these conditions all the time, but perhaps if the distribution had a large enogh time frame (as in for a week at a time), there is a likelihood that a hospital could meet these conditions. We can assume that (generally) patients enter a hospital independently of one another. The probability of a patient entering a hospital could change over time hour-by-hour, as I'd assume more people wolld go to the hospital after work if their condition is severe but not life-threatening, moreover certain dangerous behaviors (drunk driving, tired driving, etc.) tend to happen later in the day/week. However, if a unit of time was considered one week, this condition would be met since these should happen at the same rate week-by-week. We can assume that there isn't a limit to the amount of people that can enter the hospital if the hospital is well-staffed and is able to administer quick and efficient care. If we assume a constant rate from week-to-week, the proportionality condition is met.

In summary, a hospital could use the poisson distribution as an estimate if:

- They use a base time unit of one week
- They assume independence
- They assume a constant probability