In class exercise for Tutorial 13: Looping back and forward a little

Set up your GitHub Repo for the exercise

For this exercise we will ask you to go back to the beginning of the semester and use GitHub. We will also practice with loops and if statements.

To get started follow these instructions:

(1) Create a new GitHub repository under your account on GitHub.com called 'FDS-2022-Exercises'

(2) Clone the repository locally on your computer. Say under your local 'git' folder

(3) Initialize it as your like it (add a readme file, a license, etc)

(4) Create a new file inside the repository locally: tutorial13Exercise_.ipynb

(5) Add the file to the Gitreposiory

(6) Commit the to the repository and push it to the cloud.

(7) Use the above jupyter notebook and the repository you set up to complete the exercises below.

The PDF submitted on canvas of tutorial13Exercise_YourInitials.ipynb will need to report the URL of the jupyter notebook from your repository on GitHub.com. You can add the URL in the following cell.

URL of your tutorial13Exercise on GitHub:

Simulating data

Create 5 distributions of 1,000 normally distributed random data with the following characteristics:

- Dist 1: mean = 5 SD = 2
- Dist 2: mean = 7 SD = 4
- Dist 3: mean = 9 SD = 6
- Dist 4: mean = 11 SD = 8
- Dist 5: mean = 13 SD = 10

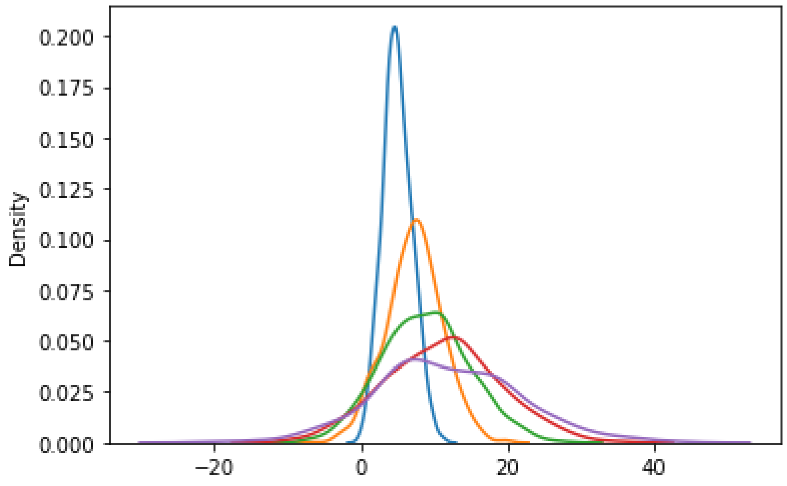
  Use for loops to generate the distributions

In [10]:
```
# import what you need for numerical arrays and pretty plots
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
import pandas as pd
```

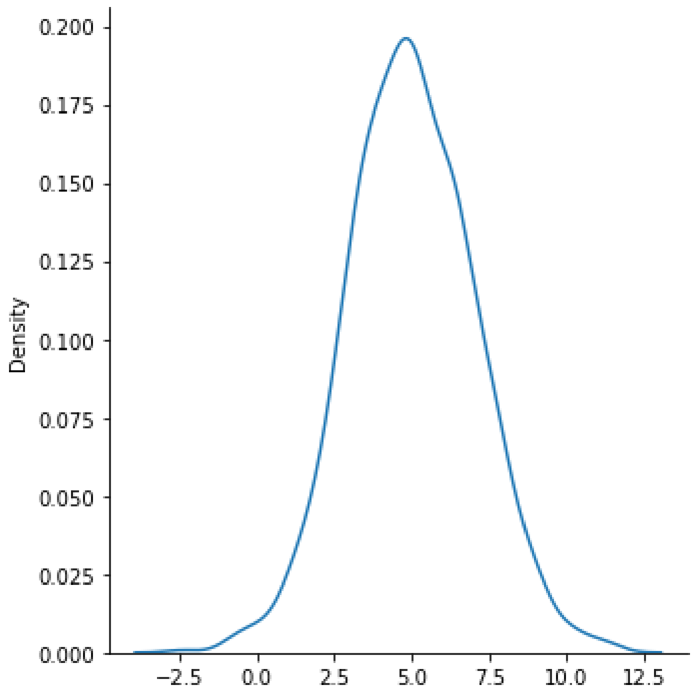
In [2]:
```
# create arrays for your means and SDs values
means = np.array([5, 7, 9, 11, 13])
SDs = np.array([2, 4, 6, 8, 10])
```

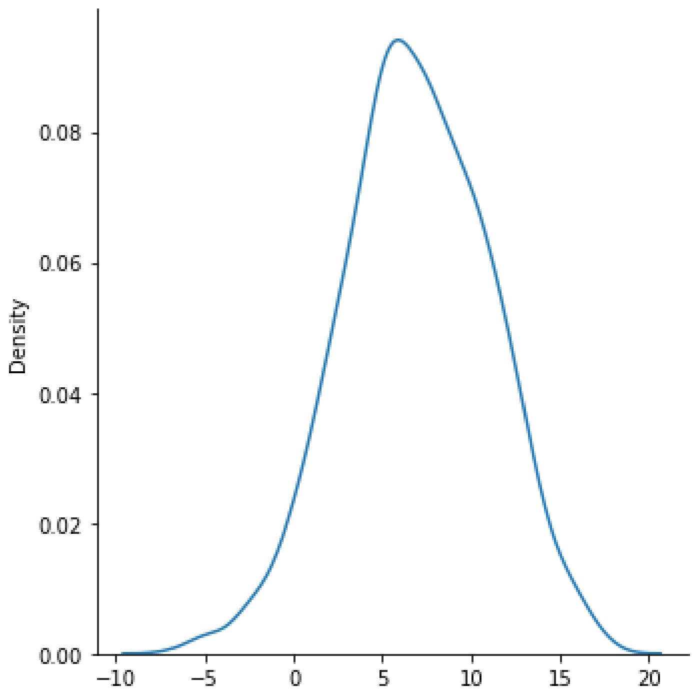
In [7]:
```
# create the distributions and store them
# into a single numpy array 'dist'
for i in range(1000):
    dist_0 = means[0] + SDs[0] * np.random.randn(1000)
    dist_1 = means[1] + SDs[1] * np.random.randn(1000)
    dist_2 = means[2] + SDs[2] * np.random.randn(1000)
    dist_3 = means[3] + SDs[3] * np.random.randn(1000)
    dist_4 = means[4] + SDs[4] * np.random.randn(1000)
arrayOfArrays = np.array([dist_0, dist_1, dist_2, dist_3, dist_4])
```

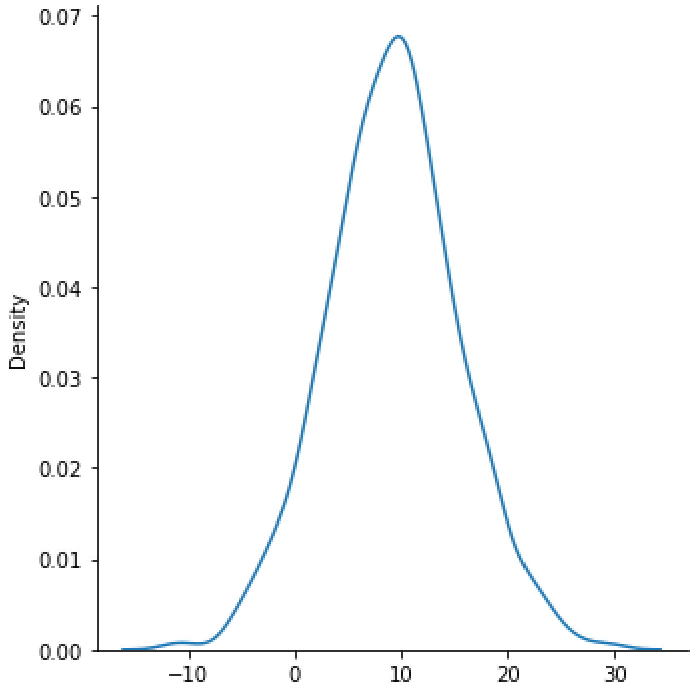
In [4]:
```
# Plot the KDE plots of the distributions in a single figure
import seaborn as sns
for i in range(5):
    sns.kdeplot(arrayOfArrays[i])
```

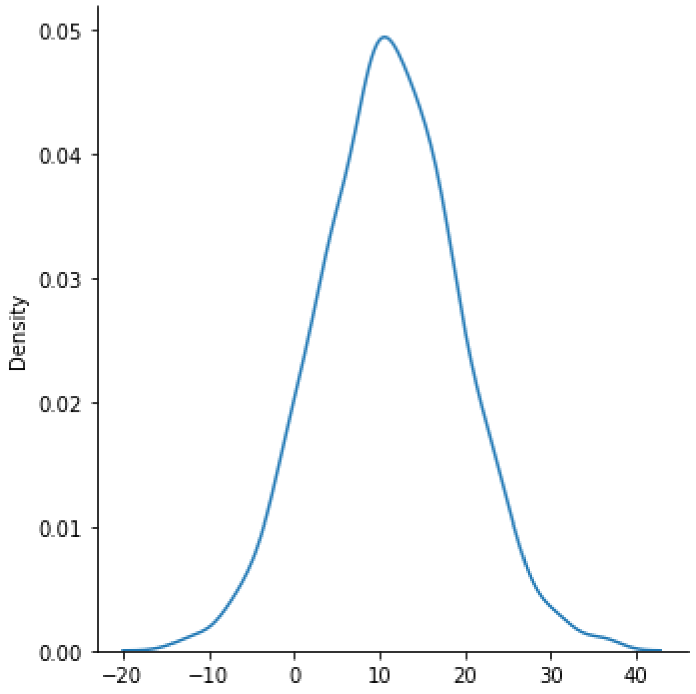


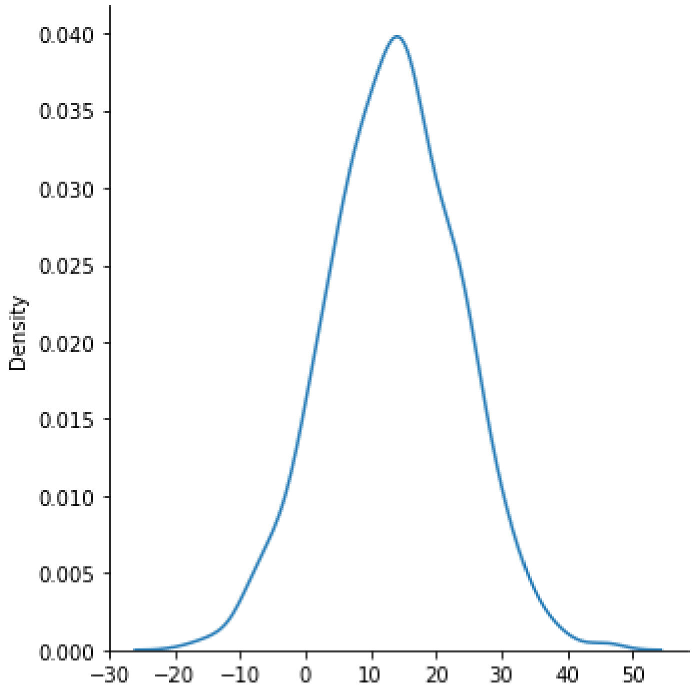
In [19]:
```
# Plot the KDE plots of the distributions in multiple subplots
#
# hint: you might need to use zip() to return two counters as a tuple
#       zip(my_index_vals, my_axis_vals)
#       https://docs.python.org/3/library/functions.html#zip
for i in range(0, 5):
    sns.displot(arrayOfArrays[i], kind="kde")
```




Density
0.00
0.02
0.04
0.06
0.08
−10
−5
0
5
10
15
20


Density
0.00
0.01
0.02
0.03
0.04
0.05
0.06
0.07
−10
0
10
20
30


Density
0.05
0.04
0.03
0.02
0.01
0.00
−20
−10
0
10
20
30
40

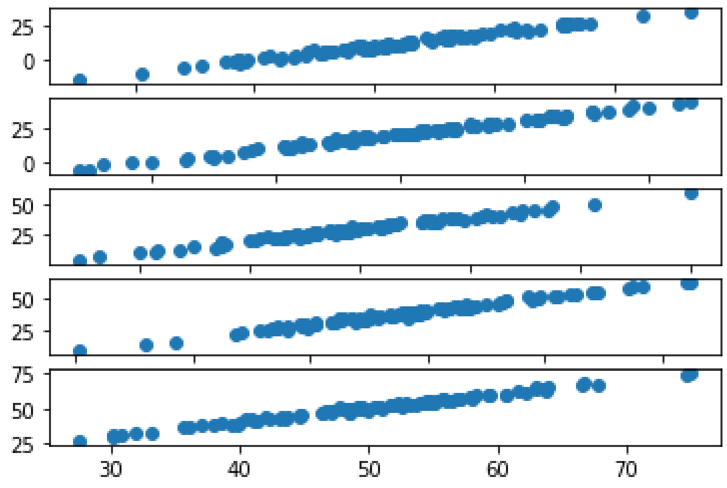

Density
0.040
0.035
0.030
0.025
0.020
0.015
0.010
0.005
0.000
−30
−20
−10
0
10
20
30
40
50

Explain why the single axis plot (first one above) and the multiple axis plot (the one using subplot) look so different. Which one is more informative and why?

Correlated scatter plots using a while loop

Create 5 different scatterplots of correlated data (fine data points per plot would be fine). To do so you can create randn. You must use a while loop to create the plots sequentially.

In [11]:
```
# the plt.pause(0.05) might be of help
fig, axs = plt.subplots(5)
counter = 1
while counter < 6:
    X = 10*counter + 10*np.random.randn(100,1)
    Y = X + np.random.randn(*X.shape)
    axs[counter-1].scatter(X,Y)
    counter += 1
```



Simulate a bunch of experimental replications using while loops

Imagine, we wanted to simulate many many repeates of the same experiments. For example, imagine that we wanted to appreciate the variability of the data obtained in the experiments, under certain conditions of noise and variability in the data.

How would we simulate a bunch of experiments? We obviously can't actually repeat the experiments in the real world. But, as data scientists, we do have a couple of options, both of which we can implement with `while` loops!

Monte Carlo Simulation

If we want to repeat the experment a bunch of times, let's consider what we know! We know that the website claims that:

- the scores are normally distributed
- they have a mean of 50
- and a standard deviation of 10

So we should be able to use `numpy.random.randn()` to generate numbers that meet the first critereon. Then we just have to scale the standard deviation up by 10 and set the mean to 50. Luckily, we know how to multiply (`*`) and add (`+`), respectively.

So here's our mission:

- write a `while` loop that repeats `n_replications = 2000` times
- on each replication
  - compute the mean of the simulated experiment
  - store that mean in a `mc_means` numpy array
- do a histogram of the means
- make a kde also too
- compute the mean and standard deviation of the 2000 means
  - compare the "mean o' means" from your simulation with the data mean
  - compare the "standard deviation o' means" with the standard error of the data

The simulation via `for` loop:

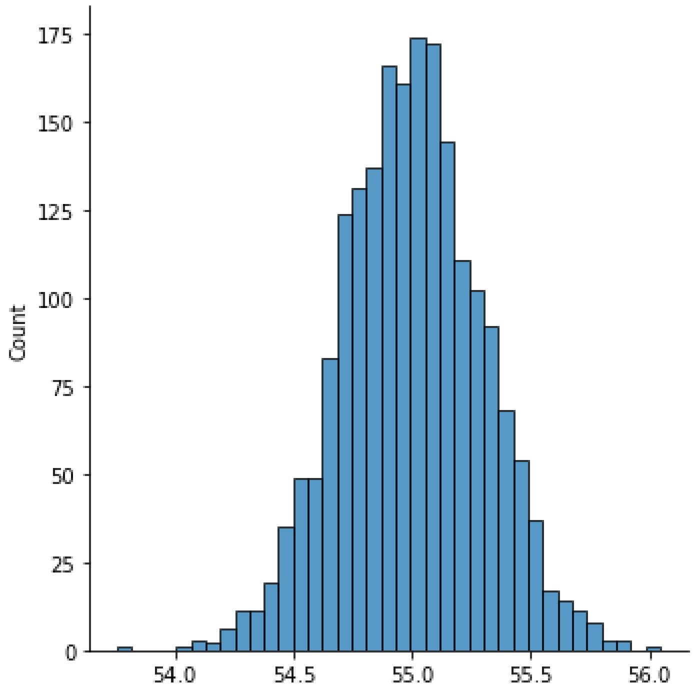
In [12]:
```
n_rep = 2000
mean_mc = 50
sd_mc = 10
n = 100
mc_means = []
sample_mean = []

for i in range (n_rep):
    sample_mean = np.random.rand(n)*sd_mc + mean_mc
    mc_means.append(sample_mean.mean())
    counter += 1
mc_means = np.array(mc_means)
mc_means.shape
```

Out[12]:
```
(2000,)
```

Histogram of the means:

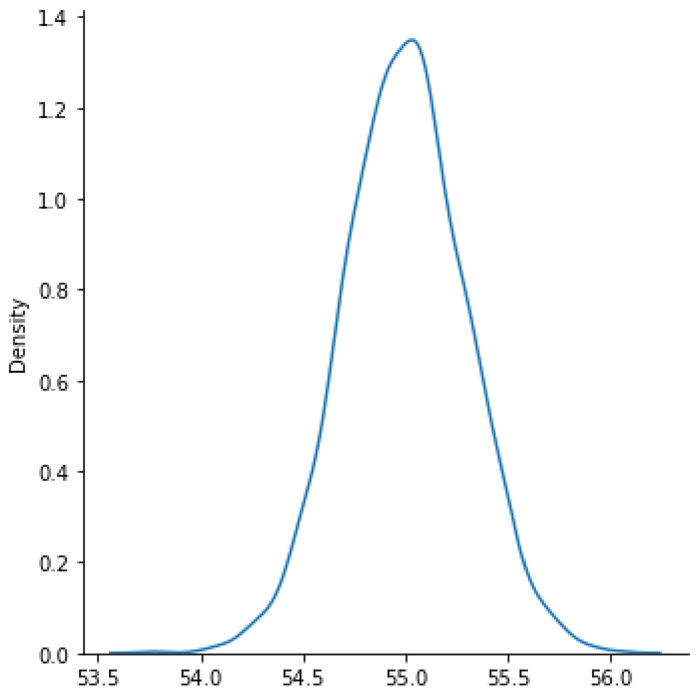
In [13]:
```
sns.displot(mc_means, kind = "hist")
```

Out[13]:
```
<seaborn.axisgrid.FacetGrid at 0x7fddeb6fe5e0>
```



KDE of the means

```
In [14]: sns.displot(mc_means, kind = "kde")

Out[14]: <seaborn.axisgrid.FacetGrid at 0x7fddda31f070>
```



Compute the mean value of your simulation means:

```
In [15]: mc_means.mean()

Out[15]: 54.99804607059875
```

Compare it with the original data mean:

```
The original data mean is approximately 50 while the simulation mean is
around 54.998.
```

Compute the standard deviation of your simulation means:

In [16]:
```
mc_means.std()
```

Out[16]:
```
0.29519828654959435
```

Compare it with the standard error you computed from the original data:

```
The standard deviation of the simulation is 0.2952 which is different
from the standard error of the 0.0979
```

Which code is easier to write and read, the one from last week that used for-loops or the one from this week that used while-loops?

```
The for loop makes easier to code since there is less variable involved.
```