K-Means Clustering

Purpose: The purpose of this lab is to learn how to use an unsupervised learning algorithm, **K-means** using sklearn.

> 1. Run a K-means algorithm.
> 2. Understand what parameters are customizable for the algorithm.
> 3. Know how to use the inertia curve to determine the optimal number of clusters.

K-Means Overview

K-means is one of the most basic clustering algorithms. It relies on finding cluster centers to group data points based on minimizing the sum of squared errors between each datapoint and its cluster center.

In [2]:
```python
# Setup and imports
# sets backend to render higher res images
%config InlineBackend.figure_formats = ['retina']
import numpy as np, pandas as pd, seaborn as sns, matplotlib.pyplot as plt
from sklearn.preprocessing import scale
from sklearn.cluster import KMeans
from sklearn.datasets import make_blobs # blobs useful for playing around with K-means.
from sklearn.utils import shuffle
```

In [5]:
```python
plt.rcParams['figure.figsize'] = [6,6]
sns.set_style("whitegrid")
sns.set_context("talk")
```

K-means clustering is one of the most simple clustering algorithms. One of the limitations is that it depends on the starting point of the clusters, and the number of clusters need to be defined beforehand.

Cluster starting points

Let's start by creating a simple dataset.

If we have a number of clusters, what we want to do is for each one of our different clusters plot out, and the way that we do this is we call plt.scatter. And then we say we want the x values, for which our K-means model came up with the label == i, whatever, that we're looping through the number of clusters here. So for the first one and the second one, and so on, then we're going to say we want that first column. And then we're going to say for all those equal to i again, and we want that second column. So we get each of the two columns, but specifying the rows that are equal to the labels that we came up with. And then we set it to different colors looping through each one of these colors that we have defined above.

And then we are also going to plot the actual cluster centers so we can see where those lies as well. So we just say cluster i related to the cluster that we are currently on. And we say the x-coordinate as well as the y-coordinate. So saying that first column and second column. And then again using that same color, and we're going to mark that with an x, so that we can differentiate that from our actual data points. We're also going to make the size of that larger, we're going to say the size is equal to 100.

In [8]:
```python
# helper function that allows us to display data in 2 dimensions an highlights the clusters
def display_cluster(X,km=[],num_clusters=0):
    color = 'brgcmyk' # Color similar to a list, We can think of it looping through b for blue, r for red, and so on.
    alpha = 0.5
    s = 20 #size
    if num_clusters == 0:
        plt.scatter(X[:,0],X[:,1],c = color[0],alpha = alpha,s = s)
    else:
        for i in range(num_clusters):
            plt.scatter(X[km.labels_==i,0],X[km.labels_==i,1],c = color[i],alpha = alpha,s=s)
            plt.scatter(km.cluster_centers_[i][0],km.cluster_centers_[i][1],c = color[i], marker = 'x', s = 100)
```

In [12]:
```python
X
```

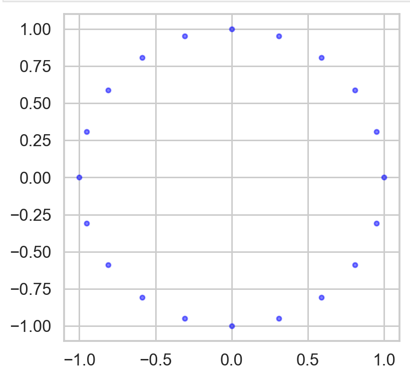
Out[12]:
```
array([[ 1.00000000e+00,  0.00000000e+00],
       [ 9.51056516e-01,  3.09016994e-01],
       [ 8.09016994e-01,  5.87785252e-01],
       [ 5.87785252e-01,  8.09016994e-01],
       [ 3.09016994e-01,  9.51056516e-01],
       [ 6.12323400e-17,  1.00000000e+00],
       [-3.09016994e-01,  9.51056516e-01],
       [-5.87785252e-01,  8.09016994e-01],
       [-8.09016994e-01,  5.87785252e-01],
       [-9.51056516e-01,  3.09016994e-01],
       [-1.00000000e+00,  1.22464680e-16],
       [-9.51056516e-01, -3.09016994e-01],
       [-8.09016994e-01, -5.87785252e-01],
       [-5.87785252e-01, -8.09016994e-01],
       [-3.09016994e-01, -9.51056516e-01],
       [-1.83697020e-16, -1.00000000e+00],
       [ 3.09016994e-01, -9.51056516e-01],
       [ 5.87785252e-01, -8.09016994e-01],
       [ 8.09016994e-01, -5.87785252e-01],
       [ 9.51056516e-01, -3.09016994e-01]])
```

To see what this looks like, we're going to create our X here. And in order to do that, we create this angle which is just going to be Numpy array, where it's values between 0 and 2 times pi. And it's going to be 20 equally spaced points. And we're saying we don't want the endpoint. So it's going to be up to but not including 2 times pi.

We're then going to append two different values together to create our X

within our x, our first feature and our second feature. So each of our two axes where the first one's going to be the cosine of our angle, and the second one's going to be the sine of our angle. And the 0 is just to say that we want to append these across the 0 axis, so that we have them one alongside the other. And then we transpose this so that we have two different columns.

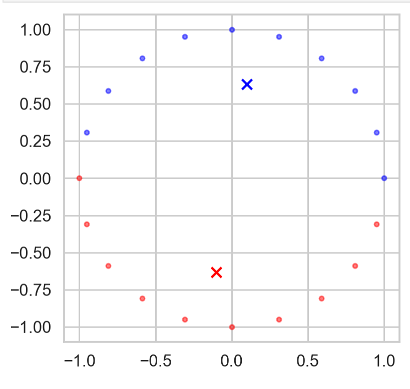
In [13]:
```python
angle = np.linspace(0,2*np.pi,20, endpoint = False)
X = np.append([np.cos(angle)],[np.sin(angle)],0).transpose()
display_cluster(X)
```



Let's now group this data into two clusters. We will use two different random states to initialize the algorithm. Settign a the **random state** variable is useful for testing and allows us to seed the randomness (so we get the same results each time).

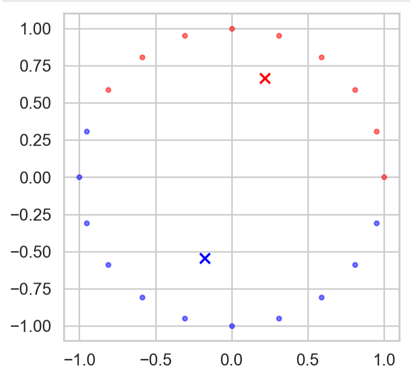
Clustering with a random state of 10:

In [14]:
```python
num_clusters = 2
km = KMeans(n_clusters=num_clusters,random_state=10,n_init=1,) # n_init, number of times the K-mean algorithm will run
km.fit(X)
display_cluster(X,km,num_clusters)
```



Clustering with a random state of 20:

In [15]:
```python
km = KMeans(n_clusters=num_clusters,random_state=20,n_init=1)
km.fit(X)
display_cluster(X,km,num_clusters)
```



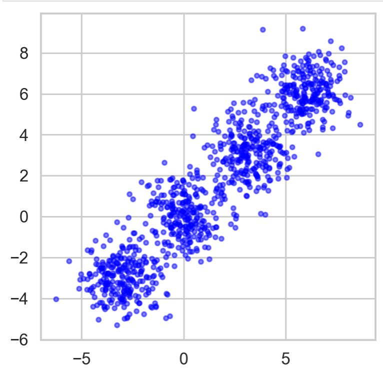
Why are the clusters different when we run the K-means twice?

It's because the starting points of the cluster centers have an impact on where the final clusters lie. The starting point of the clusters is controlled by the random state.

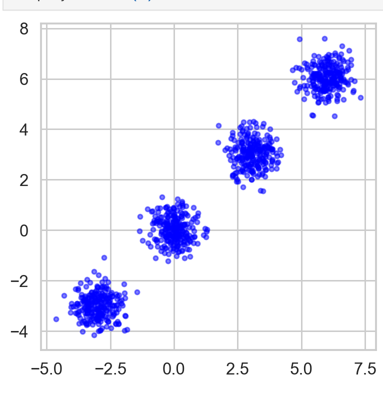
Determining optimium number of clusters

Let's create a new dataset that visually consists on a few clusters and try to group them.

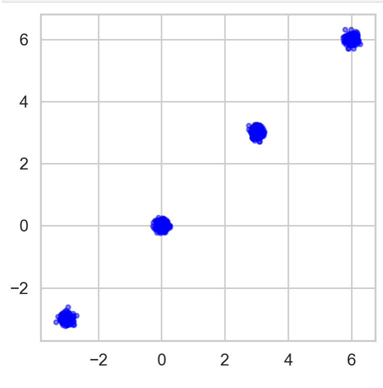
In [16]:
```python
n_samples = 1000
n_bins = 4
centers = [(-3, -3), (0, 0), (3, 3), (6, 6)]
X, y = make_blobs(n_samples=n_samples, n_features=2, cluster_std=1.0, #std will define how tightly around each one of these centroids are going to plot each one of our data points.
                  centers=centers, shuffle=False, random_state=42)
display_cluster(X)
```



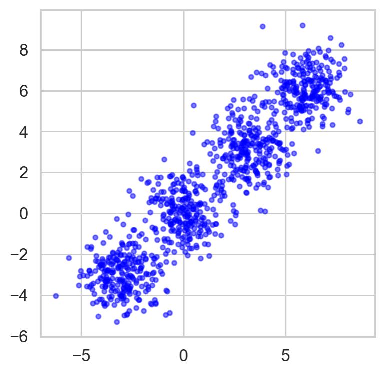
In [19]:
```python
n_samples = 1000
n_bins = 4
centers = [(-3, -3), (0, 0), (3, 3), (6, 6)]
X, y = make_blobs(n_samples=n_samples, n_features=2, cluster_std=0.5, # if smaller standard deviation,they are really tight around each cluster
                  centers=centers, shuffle=False, random_state=42)
display_cluster(X)
```



In [27]:
```python
n_samples = 1000
n_bins = 4
centers = [(-3, -3), (0, 0), (3, 3), (6, 6)]
X, y = make_blobs(n_samples=n_samples, n_features=2, cluster_std=0.10, # if smaller standard deviation,they are really tight around each cluster
                  centers=centers, shuffle=False, random_state=42)
display_cluster(X)
```

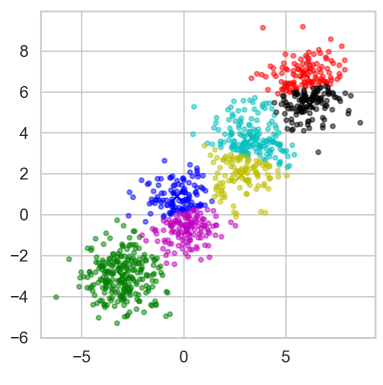


In [28]:
```python
n_samples = 1000
n_bins = 4
centers = [(-3, -3), (0, 0), (3, 3), (6, 6)]
X, y = make_blobs(n_samples=n_samples, n_features=2, cluster_std=1.0, # if smaller standard deviation,they are really tight around each cluster
                  centers=centers, shuffle=False, random_state=42)
display_cluster(X)
```



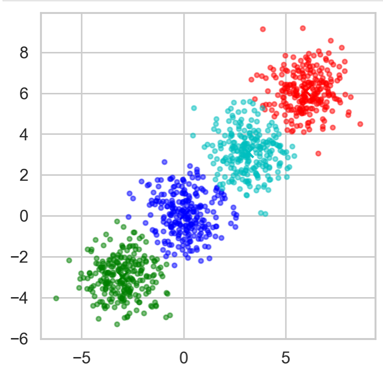
Let's run K-means with seven clusters.

In [29]:
```python
num_clusters = 7
km = KMeans(n_clusters=num_clusters)
km.fit(X)
display_cluster(X,km,num_clusters)
```



Now let's re-run the algorithm with four clusters.

In [30]:
```python
num_clusters = 4
km = KMeans(n_clusters=num_clusters)
km.fit(X)
display_cluster(X,km,num_clusters)
```



Should we use four or seven clusters?

- In this case it may be visually obvious that four clusters is better than seven.
- This is because we can easily view the data in two dimensional space.
- However, real world data usually has more than two dimensions.
- A dataset with a higher dimensional space is hard to visualize.
- A way of solving this is to plot the **inertia**

inertia: (sum of squared error between each point and its cluster center) as a function of the number of clusters.

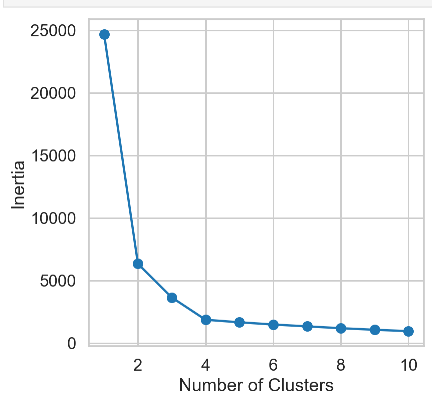
In [31]:
```python
km.inertia_
```

Out[31]:
```
1880.216477378221
```

create a blank list. We have inertia = []. we're going to run through a range of different numbers of clusters ranging from 1-11, including 11-10. Then for num clusters in this list, so for that 1-10, we're going to fit a k-means on that number of clusters. We're then going to take that inertia list and append on for our fitted model, We're then going to plot as our x axis. We're going to use the list num clusters, which is going to be those values 1-10, and as our y-axis is going to be these inertia values that were coming up with that we're appending onto the list. We call plt.scatter on these two. So this will actually create a line plot, this will create our actual markers. We're then going to set our x-label and our y-label to number of clusters and inertia respectively,

In [32]:
```python
### BEGIN SOLUTION
inertia = []
list_num_clusters = list(range(1,11))
for num_clusters in list_num_clusters:
    km = KMeans(n_clusters=num_clusters)
    km.fit(X)
    inertia.append(km.inertia_) #Inertia for that given model for clusters equal to 1, 2, 3, etc, up until 10.

plt.plot(list_num_clusters,inertia) # line plot
plt.scatter(list_num_clusters,inertia) #create markers
plt.xlabel('Number of Clusters')
plt.ylabel('Inertia');
### END SOLUTION
```



Where does the elbow of the curve occur?

What do you think the inertia would be if you have the same number of clusters and data points?

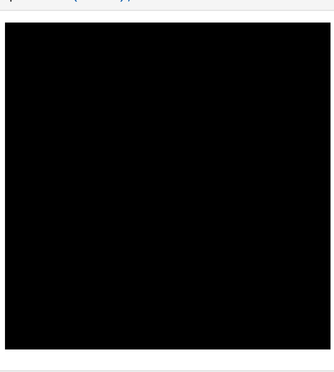
Clustering Colors

Each pixel has 3 values that represent how much red, green and blue it has. Below you can play with different combinations of RGB to create different colors. In total, you can create $256^3 = 16,777,216$ unique colors.

In [34]:
```python
# assign values for the RGB.  Each value should be between 0 and 255
R = 35
G = 95
B = 131
plt.imshow([[np.array([R,G,B]).astype('uint8')]])
plt.axis('off');
```

In [35]:
```python
# assign values for the RGB.  Each value should be between 0 and 255
R = 35
G = 95
B = 90
plt.imshow([[np.array([R,G,B]).astype('uint8')]])
plt.axis('off');
```

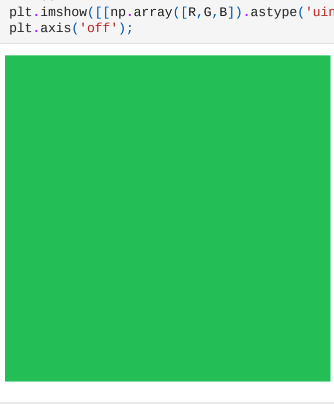
In [36]:
```python
# assign values for the RGB.  Each value should be between 0 and 255
R = 0
G = 0
B = 0
plt.imshow([[np.array([R,G,B]).astype('uint8')]])
plt.axis('off');
```



In [37]:
```python
# assign values for the RGB.  Each value should be between 0 and 255
R = 255
G = 255
B = 255
plt.imshow([[np.array([R,G,B]).astype('uint8')]])
plt.axis('off');
#white it is
```

In [38]:
```python
# assign values for the RGB.  Each value should be between 0 and 255
R = 180
G = 62
B = 31
plt.imshow([[np.array([R,G,B]).astype('uint8')]])
plt.axis('off');
```

In [39]:
```python
# assign values for the RGB.  Each value should be between 0 and 255
R = 35
G = 190
B = 86
plt.imshow([[np.array([R,G,B]).astype('uint8')]])
plt.axis('off');
```



In [40]:
```python
# assign values for the RGB.  Each value should be between 0 and 255
R = 125
G = 125
B = 125
plt.imshow([[np.array([R,G,B]).astype('uint8')]])
plt.axis('off');
```