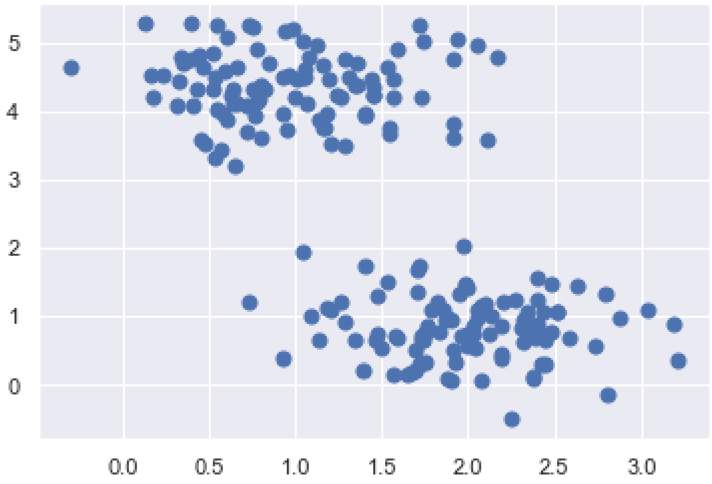
In [1]:
```
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
sns.set()
```

Consider a small "city" of people.

- Each point represents a person
- Friendships are formed entirely based on how close they live to each other

Could you put these people into communities?

In [2]:
```
from sklearn.datasets.samples_generator import make_blobs
X, y_true = make_blobs(n_samples=200, centers=2,
                       cluster_std=0.50, random_state=0)
plt.scatter(X[:, 0], X[:, 1], s=50);
```



How would you tell a program to do what you did visually?

Remember how the computer "sees" these points

In [3]:

```
# Print first 15 points
print(X[:15, :])
```

```
[[2.43859911 1.07581007]
 [1.85554301 1.0826916 ]
 [2.58952222 0.67097076]
 [1.73654901 0.69902775]
 [1.74265969 5.03846671]
 [0.64003985 4.12401075]
 [1.04829186 5.03092408]
 [0.5323772  3.31338909]
 [1.98882723 0.74876822]
 [0.16117091 4.53517846]
 [1.7571105  0.87138001]
 [1.28486901 0.92929466]
 [1.16448284 3.75408693]
 [0.3498724  4.69253251]
 [2.10413001 1.1891405 ]]
```

Brief aside on multi-dimensional numpy arrays

- Shape
- Indexing
- Boolean indexing
- Slicing

In [4]:

```
# Shape
print('The shape of the array is %s' % (str(X.shape)))
print('The number of samples is %d' % X.shape[0])
print('The number of dimensions is %d' % X.shape[1])
```

```
The shape of the array is (200, 2)
The number of samples is 200
The number of dimensions is 2
```

In [5]:

```
# Indexing
i = 4
j = 1
print('The %d-th dimension of sample %d is %g' % (j+1, i+1, X[i, j]))

temp = -10*np.arange(10) # numbers 0-9
print('Selecting 1st 3rd and 8th part of array')
print(temp[[1,3,8]])
```

```
The 2-th dimension of sample 5 is 5.03847
Selecting 1st 3rd and 8th part of array
[-10 -30 -80]
```

In [6]:

```
# Boolean indexing
temp = -10*np.arange(10) # numbers 0-9
selection = np.zeros(temp.shape[0], dtype=bool)
print('Boolean array')
print(selection)
selection[1] = True
selection[3] = True
selection[8] = True
# Or equivalently selection[[1,3,8]] = True

print('Selecting 1st 3rd and 8th part of array via boolean array')
print(temp[selection])
```

```
Boolean array
[False False False False False False False False False False]
Selecting 1st 3rd and 8th part of array via boolean array
[-10 -30 -80]
```

In [7]:

```
# Slicing
print('The first 10 samples with all dimensions')
print(X[:10, :])
```

```
The first 10 samples with all dimensions
[[2.43859911 1.07581007]
 [1.85554301 1.0826916 ]
 [2.58952222 0.67097076]
 [1.73654901 0.69902775]
 [1.74265969 5.03846671]
 [0.64003985 4.12401075]
 [1.04829186 5.03092408]
 [0.5323772  3.31338909]
 [1.98882723 0.74876822]
 [0.16117091 4.53517846]]
```

In [8]:

```
print('The %d-th sample is:' % (i+1))
print(str(X[i, :]))
```

```
The 5-th sample is:
[1.74265969 5.03846671]
```

In [9]:

```
print('The first 10 samples for the dimension %d' % (j+1))
print(X[:10, j])
```

```
The first 10 samples for the dimension 2
[1.07581007 1.0826916  0.67097076 0.69902775 5.03846671 4.12401075
 5.03092408 3.31338909 0.74876822 4.53517846]
```

In [10]:

```
print('The last 10 samples for the dimension %d' % (j+1))
print(X[-10:, j])
```

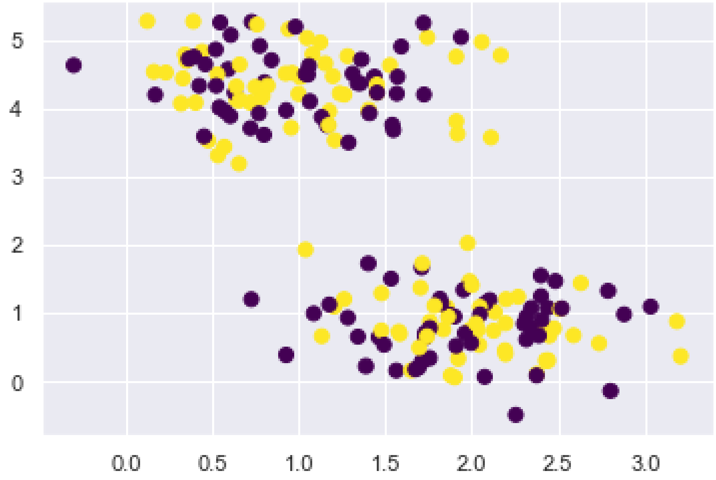
```
The last 10 samples for the dimension 2
[1.10568868 3.97204818 1.12313089 0.84858847 1.46821459 1.24503823
 4.2012082  3.57660449 4.22810872 4.75420057]
```

How do we formalize what we did visually?

- Let's assume for now that we know there are exactly *two* communities
- How can we assign each person to a community?
- Naive idea: Randomly assign points to each community

In [11]:
```
from sklearn.utils import check_random_state
def get_random_assignment(random_state=None):
    rng = check_random_state(random_state)
    y = rng.randint(2, size=X.shape[0])
    return y
y_rand = get_random_assignment(random_state=0)
plt.scatter(X[:, 0], X[:, 1], c=y_rand, s=50, cmap='viridis')
```

Out[11]:
```
<matplotlib.collections.PathCollection at 0x1a1f8777f0>
```



This clustering "looks" quite bad.

How can we formalize whether a particular assignment is good or bad?

- One intuition: People in a communities will be as close to each other as possible.
- Take average distance between each person in a community to every other person in the **same** community.
- Sum over all communities.

(Derive on board)

Implement (Euclidean) distance function

$$\text{dist}(x, z) = \sqrt{(x_1 - z_1)^2 + (x_2 - z_2)^2}$$

In [12]:

```python
# Euclidean distance function
def distance(xvec, zvec, show=False):
    diff = xvec - zvec
    squared = diff * diff
    sum_squared = np.sum(squared)
    d = np.sqrt(sum_squared)
    if show:
        print('xvec', xvec)
        print('zvec', zvec)
        print('diff', diff)
        print('squared', squared)
        print('sum_squared', sum_squared)
        print('distance', d)
    return d

print('Simple')
distance(np.array([0, 1]), np.array([1, 0]), show=True)
print()
print('Real example')
distance(X[0, :], X[1, :], show=True)
```

```
Simple
xvec [0 1]
zvec [1 0]
diff [-1  1]
squared [1 1]
sum_squared 2
distance 1.4142135623730951

Real example
xvec [2.43859911 1.07581007]
zvec [1.85554301 1.0826916 ]
diff [ 0.58305611 -0.00688154]
squared [3.39954422e-01 4.73555265e-05]
sum_squared 0.3400017776337018
distance 0.5830967137908615
```

Out[12]:

```
0.5830967137908615
```

Programming: `zip(a,b)` - Your looping friend, generally much better than indices like `i` or `j` if possible

In [31]:
```
num_arr = 10*np.arange(10)
char_list = ['a','b','c','d']

for n, c in zip(num_arr, char_list):
    print('n=', n, 'c=', c)

for n, c, xvec in zip(num_arr, char_list, X):
    print('n=', n, 'c=', c, 'xvec=', xvec)
```

```
n= 0 c= a
n= 10 c= b
n= 20 c= c
n= 30 c= d
n= 0 c= a xvec= [2.43859911 1.07581007]
n= 10 c= b xvec= [1.85554301 1.0826916 ]
n= 20 c= c xvec= [2.58952222 0.67097076]
n= 30 c= d xvec= [1.73654901 0.69902775]
```

In [35]:
```
# Loop through first 10 x-y pairs
for xvec, yy in zip(X[:10], y_true[:10]):
    print('xvec=', xvec, 'y=', yy)
```

```
xvec= [2.43859911 1.07581007] y= 1
xvec= [1.85554301 1.0826916 ] y= 1
xvec= [2.58952222 0.67097076] y= 1
xvec= [1.73654901 0.69902775] y= 1
xvec= [1.74265969 5.03846671] y= 0
xvec= [0.64003985 4.12401075] y= 0
xvec= [1.04829186 5.03092408] y= 0
xvec= [0.5323772  3.31338909] y= 0
xvec= [1.98882723 0.74876822] y= 1
xvec= [0.16117091 4.53517846] y= 0
```

In [29]:
```
# Loop through columns of data matrix
for xcol in X.transpose():
    print(xcol[:10]) # Only show first 10 elements of column
```

```
[2.43859911 1.85554301 2.58952222 1.73654901 1.74265969 0.64003985
 1.04829186 0.5323772  1.98882723 0.16117091]
[1.07581007 1.0826916  0.67097076 0.69902775 5.03846671 4.12401075
 5.03092408 3.31338909 0.74876822 4.53517846]
```

Note: `zip` will only match elements up to the shortest iterable

Implement objective with loop

$$\mathcal{C}_j = \{x \in \mathcal{X} : y = j\}$$

$$\sum_{j=1}^{k} \frac{1}{2|\mathcal{C}_j|} \sum_{x \in \mathcal{C}_j, z \in \mathcal{C}_j} \operatorname{dist}(x, z)^2$$

In [14]:

```
def objective_loop(X, y):
    k = len(np.unique(y))
    out = 0
    for j in range(k):
        n_community = 0
        community_sum = 0
        for xvec, y1 in zip(X, y):
            if y1 != j:
                continue
            n_community += 1
            for zvec, y2 in zip(X, y):
                if y2 != j:
                    continue
                dist = distance(xvec, zvec)
                community_sum += dist**2
        out += community_sum / (2*n_community)
    return out

print(objective_loop(X, y_rand))
```

```
767.2572924351306
```

Implement objective in via vectorized calls

$$\mathcal{C}_j = \{x \in \mathcal{X} : y = j\}$$

$$\sum_{j=1}^{k} \frac{1}{2|\mathcal{C}_j|} \sum_{x \in \mathcal{C}_j, z \in \mathcal{C}_j} \mathrm{dist}(x, z)^2$$

In [15]:

```
from sklearn.metrics import pairwise_distances
# Using vectorized computation
def objective(X, y):
    y_vals = np.unique(y)
    out = 0
    for yv in y_vals:
        sel = (y==yv)   # boolean array
        Xj = X[sel, :]
        n_community = np.sum(sel)
        community_sum = np.sum(pairwise_distances(Xj, Xj)**2)
        out += community_sum / (2*n_community)
    return out

print(objective(X, y_rand))
print('Difference from loop version = %g' % (objective(X, y_rand) - obje
ctive_loop(X, y_rand)))
```

```
767.2572924351311
Difference from loop version = 5.68434e-13
```

Programming: *List Comprehensions*, also your friend, MAP and FILTER operations

- Suppose you want to map a list of numbers to a list of strings
- Suppose you only want to map odd numbers
- Syntax `[<expression> for <item> in <iterable> if <condition>]`

In [16]:

```
num_list = [1, 2, 3, 4, 5]
str_list = []
for i in range(len(num_list)):
    str_list.append('Num:' + str(num_list[i]))
print('With index loops')
print(str_list)

# Use iterators
str_list = []
for n in num_list:
    str_list.append('Num:' + str(n))
print('With iterator loops')
print(str_list)

# Use list comprehension
str_list = ['Num:' + str(n) for n in num_list]
print('With list comprehension')
print(str_list)

# Use list comprehension
str_list = ['Num:' + str(n) for n in num_list if np.mod(n, 2) == 1]
print('Odd numbers only with list comprehension')
print(str_list)
```

```
With index loops
['Num:1', 'Num:2', 'Num:3', 'Num:4', 'Num:5']
With iterator loops
['Num:1', 'Num:2', 'Num:3', 'Num:4', 'Num:5']
With list comprehension
['Num:1', 'Num:2', 'Num:3', 'Num:4', 'Num:5']
Odd numbers only with list comprehension
['Num:1', 'Num:3', 'Num:5']
```

In [17]:
```
from sklearn.metrics import pairwise_distances
# Using vectorized and list comprehensions computation
def objective(X, y):
    y_vals = np.unique(y)
    def inner(yv):
        sel = (y==yv)  # boolean array
        Xj = X[sel, :]
        n_community = np.sum(sel)
        community_sum = np.sum(pairwise_distances(Xj, Xj)**2)
        return community_sum / (2*n_community)
    return np.sum([inner(yv) for yv in y_vals])

print(objective(X, y_rand))
print('Difference from loop version = %g' % (objective(X, y_rand) - obje
ctive_loop(X, y_rand)))
```

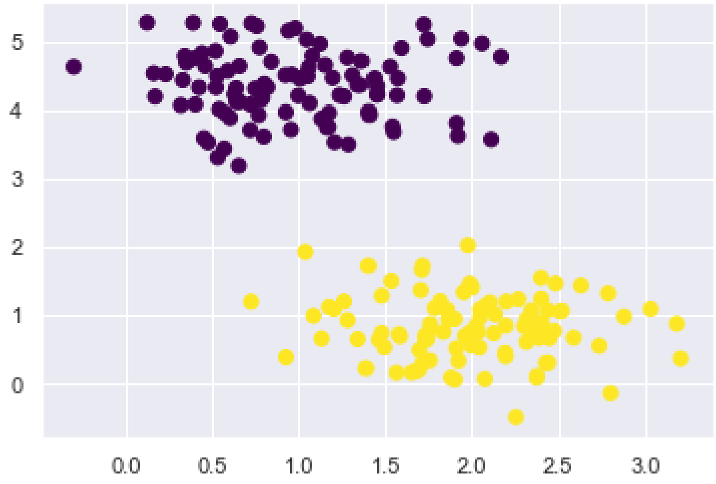
```
767.2572924351311
Difference from loop version = 5.68434e-13
```

Intuition sanity check, does visual clustering solution have a low value?

In [18]:
```
print(objective(X, y_true))
plt.scatter(X[:, 0], X[:, 1], c=y_true, s=50, cmap='viridis')
```

```
94.67363954089785
```

Out[18]:
```
<matplotlib.collections.PathCollection at 0x1a1f9e2710>
```



Clustering goal: Minimize objective over possible community assignments

$$\arg\min_{\mathcal{C}_1,\mathcal{C}_2} \sum_{j=1}^{k} \frac{1}{2|\mathcal{C}_j|} \sum_{x\in\mathcal{C}_j, z\in\mathcal{C}_j} \mathrm{dist}(x,z)^2$$

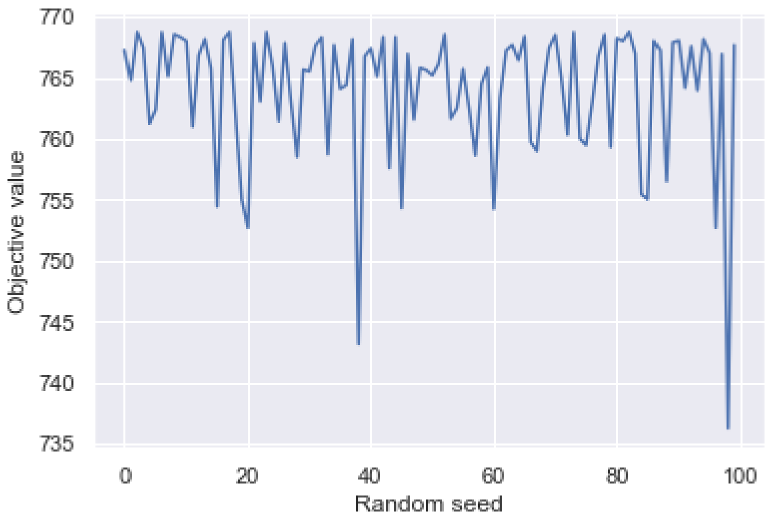
- Naively, we could just enumerate all possibilities
- Let's try several random combinations

In [19]:

```
rand_obj = np.nan * np.ones(100)
for seed in range(rand_obj.shape[0]):
    y_rand = get_random_assignment(random_state=seed)
    rand_obj[seed] = objective(X, y_rand)
    #print('Seed = %2d, Objective = %g' % (seed, obj))
plt.plot(rand_obj)
plt.xlabel('Random seed')
plt.ylabel('Objective value')
#plt.scatter(X[:, 0], X[:, 1], c=y_rand, s=50, cmap='viridis')
```

Out[19]:

```
Text(0, 0.5, 'Objective value')
```



How many possible assignments are there?

In terms of the number of samples n and the number of communities k