In [1]:
```
%matplotlib inline
import math
import time
import numpy as np
import torch
from d2l import torch as d2l
```

In [2]:
```
n = 10000
a = torch.ones(n)
b = torch.ones(n)
```

In [3]:
```
c = torch.zeros(n)
t = time.time()
for i in range(n):
    c[i] = a[i] + b[i]
f'{time.time() - t:.5f} sec'
```

Out[3]:
```
'0.08976 sec'
```

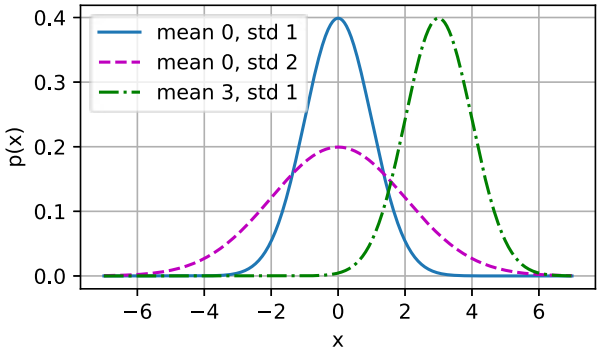
In [4]:
```
t = time.time()
d = a + b
f'{time.time() - t:.5f} sec'
```

Out[4]:
```
'0.00038 sec'
```

In [5]:
```
def normal(x, mu, sigma):
    p = 1 / math.sqrt(2 * math.pi * sigma**2)
    return p * np.exp(-0.5 * (x - mu)**2 / sigma**2)
```

In [6]:
```
x = np.arange(-7, 7, 0.01)

params = [(0, 1), (0, 2), (3, 1)]
d2l.plot(x, [normal(x, mu, sigma) for mu, sigma in params], xlabel='x',
         ylabel='p(x)', figsize=(4.5, 2.5),
         legend=[f'mean {mu}, std {sigma}' for mu, sigma in params])
```



Discussion: The content covered the basic concepts of linear regression, what a loss function is, and how to find the optimal weights through stochastic gradient descent. I came to understand that the ultimate goal is to adjust each parameter in such a way as to minimize the value of the loss function. In the coding part, what surprised me the most was that performing addition element-wise versus adding an entire vector at once with the + operator showed a significant difference in time, with the latter being much faster. Honestly, since the result is also a vector, I still don't quite understand why adding it all at once is so much faster, considering that each element needs to be summed anyway. Additionally, we covered topics on normal distribution and squared loss, and thankfully, I was already somewhat familiar with these concepts.