Discrete Bayes Filter

卡尔曼滤波器属于贝叶斯滤波器家族。大多数教科书中关于卡尔曼滤波器的处理都给出了贝叶斯公式，也许显示了它是如何融入卡尔曼滤波器方程的，但大多数情况下都将讨论保持在一个非常抽象的水平上。

这种方法需要对几个数学领域有相当复杂的理解，它仍然留给读者很多理解和形成对情况的直观把握的工作。

我将使用一种不同的方式来展开这个话题，这要归功于 Dieter Fox 和 Sebastian Thrun 的工作。这取决于通过追踪一个穿过走廊的物体来建立对贝叶斯统计如何工作的直觉——他们用机器人，我用狗。我喜欢狗，它们比机器人更难预测，这给过滤带来了有趣的困难。我能找到的第一个公开的例子似乎是 Fox 1999[1]，更完整的例子是 Fox 2003[2]。Sebastian Thrun 在他的 Udacity 课程《人工智能机器人[3]》中也使用了这个公式。事实上，如果你喜欢看视频，我强烈建议你暂停阅读这本书，先学习这门课的前几节课，然后再回到这本书，更深入地研究这个话题。

现在让我们使用一个简单的思想实验，就像我们使用 g-h 过滤器一样，来看看我们如何使用概率进行过滤和跟踪。

Tracking a Dog

让我们从一个简单的问题开始。我们有一个对狗狗友好的工作环境，所以人们带着他们的狗狗去工作。狗偶尔会走出办公室，在大厅里溜达。我们希望能够追踪到他们。在一次黑客马拉松中，有人发明了一种声呐传感器，安装在狗的项圈上。它会发出信号，听回声，根据回声返回的速度，我们可以判断狗狗是否在敞开的门口。当狗走的时候，它也能感觉到，并报告狗朝哪个方向移动。它通过 wifi 连接到网络，每秒钟发送一次更新。

我想追踪我的狗西蒙，所以我把设备绑在它的项圈上然后启动 Python，准备编写代码在整栋楼里追踪它。乍一看，这似乎是不可能的。如果我开始监听西蒙项圈的传感器，我可能会读到 door, hall, hall 等等。我要怎么用这些信息来确定西蒙的位置?

为了使问题足够小，便于绘图，我们将假设走廊上只有 10 个位置，我们将其编号为 0 到 9，其中 1 在 0 的右边。稍后我们会说明原因，我们也会假设走廊是圆形或长方形的。如果你从 9 号位置向右移动，你会在 0 号位置。

当我开始监听传感器时，我没有理由相信西蒙在走廊的任何特定位置。在我看来，他处于任何位置的可能性都是一样的。一共有 10 个位置，所以他在任意位置的概率是 1/10。

让我们表示他在 NumPy 数组中的位置。我可以使用 Python 列表，但是 NumPy 数组提供的功能我们很快就会用到。

In [3]:
```
import numpy as np
belief = np.array([1/10]*10)
print(belief)
```

```
[0.1 0.1 0.1 0.1 0.1 0.1 0.1 0.1 0.1 0.1]
```

在贝叶斯统计中，这被称为先验。它是在纳入测量值或其他信息之前的概率。更确切地说，这叫做先验概率分布。概率分布是一个事件的所有可能概率的集合。概率分布的和总是 1 因为总有事情发生;该分布列出了所有可能的事件和每个事件的概率。

我相信你以前用过概率——比如“今天下雨的概率是 30%”。最后一段听起来更像这样。但贝叶斯统计是概率领域的一场革命，因为它将概率视为对单个事件的一种信任。我们举个例子。我知道如果我把一枚均匀的硬币抛无限次，我会得到 50%的正面和 50%的反面。这被称为频率主义统计，以区别于贝叶斯统计。计算是基于事件发生的频率。

我再抛一次硬币，让它落在地上。我觉得它是往哪落的?频率论概率与此无关;它仅仅说明 50%的硬币抛掷是正面朝上。在某种程度上，给硬币当前状态赋一个概率是没有意义的。不是正面就是反面，我们只是不知道是哪个。贝叶斯认为这是一种对单一事件的信任——我的信任或知识的强度是 50%，即这枚硬币是正面的。有些人反对“信任”这个词;信任可以指在没有证据的情况下认为某事是真的。在这本书中，它总是衡量我们知识的力量。我们会在接下来的过程中了解更多。

贝叶斯统计考虑到过去的信息(先前的)。我们观察到每 100 天下雨 4 次。由此可知，明天下雨的概率是 1/25。天气预报可不是这么做的。如果我知道今天在下雨，而风暴锋停滞了，那么明天很可能会下雨。天气预报是贝叶斯的。

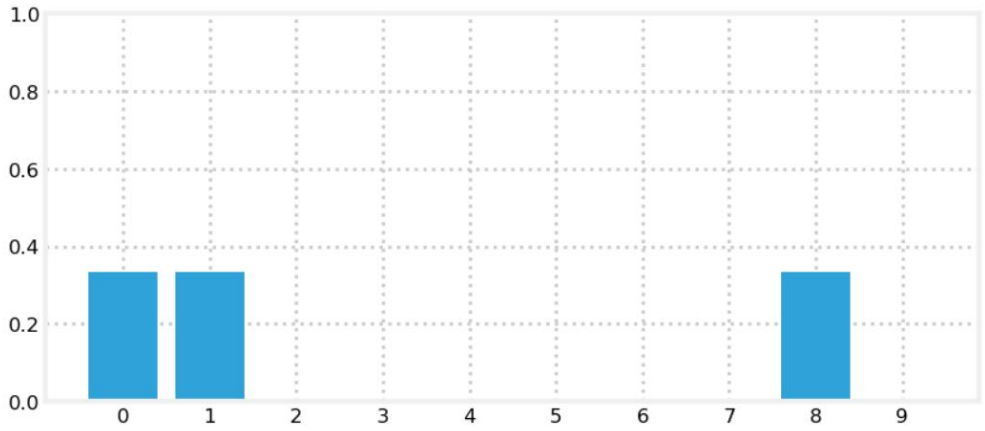
在实践中，统计学家混合使用频率主义者和贝叶斯技术。有时找到先验是困难的或不可能的，而频率主义技术是规则。在这本书中我们可以找到先验。当我说到某件事的概率时，我指的是根据过去的事件，某件事是正确的概率。我用的是贝叶斯方法。

现在让我们创建一个走廊的地图。我们将把前两扇门关在一起，然后把另一扇门放在更远的地方。我们将 1 用于门，0 用于墙:

```
In [4]: hallway = np.array([1, 1, 0, 0, 0, 0, 0, 0, 1, 0])
```

我开始监听西蒙在网络上的传输，我从传感器得到的第一个数据是门。目前假设传感器总是返回正确的答案。由此我推断他是在一扇门前，但是是哪一扇?我没有理由相信他在第一、第二、第三扇门前。我能做的就是给每扇门分布一个概率。所有的门都是等概率的，一共有三扇门，所以每扇门的概率都是 1/3。

```
In [5]: import kf_book.book_plots as book_plots
from kf_book.book_plots import figsize, set_figsize
import matplotlib.pyplot as plt

belief = np.array([1/3, 1/3, 0, 0, 0, 0, 0, 0, 1/3, 0])
book_plots.bar_plot(belief)
```



这种分布被称为分类分布，它是描述观察 n 种结果概率的离散分布。这是一个多模态分布，因为我们对狗的位置有多种看法。当然，我们并不是说我们认为他同时在三个不同的地点，只是说我们已经把我们的知识范围缩小到这三个地点中的一个。我的(贝叶斯)信任是有 33.3%的概率出现在 0 号门，33.3%出现在 1 号门，33.3%出现在 8 号门。

这是两方面的改进。我已经拒绝了一些走廊的位置，因为不可能，我对剩下的位置的信任从 10%增加到 33%。这总是会发生的。随着我们知识的提高，概率会接近 100%。

关于分布方式，我简单说几句。给定一个数字列表，例如{1,2,2,2,3,3,4}，众数是出现频率最高的数字。对于这个设置，模式是 2。一个发行版可以包含多个模式。列表{1,2,2,2,3,3,4,4,4}包含模式 2 和 4，因为两者都出现了 3 次。我们说前一个列表是单峰的，后一个列表是多峰的。

这种分布的另一个术语是直方图。直方图以图形的方式描述了一组数字的分布。上面的条形图是一个直方图。

我在上面的代码中手工编写了信任度数组。我们如何在代码中实现它?我们用 1 表示门，用 0 表示墙，所以我们将走廊变量乘以百分比，就像这样：

```
In [6]: belief = hallway * (1/3)
        print(belief)
```

```
[0.333 0.333 0.    0.    0.    0.    0.    0.    0.333 0.   ]
```

Extracting Information from Sensor Readings

让我们把 Python 放在一边，稍微考虑一下这个问题。假设我们从西蒙的传感器读取以下内容:

门

向右移动

门

我们能推断西蒙的位置吗?当然!考虑到走廊的布局，你只能从一个地方获得这个序列，那就是左侧末端。因此，我们可以自信地说，西蒙在第二道门的前面。如果这还不清楚，假设西蒙是从第二或第三扇门开始的。向右移动后，他的传感器会返回“墙”。这和传感器读数不符，所以我们知道他不是从那开始的。我们可以对所有剩下的起始位置继续这个逻辑。唯一的可能是他现在在第二扇门前。我们的信任是:

```
In [7]: belief = np.array([0., 1., 0., 0., 0., 0., 0., 0., 0., 0.])
```

我设计了走廊的布局和传感器读数来快速给出准确的答案。真正的问题并不那么明确。但这应该会激发你的直觉——第一个传感器读数只给了我们西蒙位置的低概率(0.333)，但在位置更新和另一个传感器读数之后，我们知道了更多他的位置。你可能会怀疑，如果你有一个很长的走廊，里面有大量的门，那么经过几次传感器读数和位置更新之后，我们要么就能知道西蒙在哪里，要么就能把可能性缩小到少数几种。当一组传感器读数只匹配一个到几个起始位置时，这是可能的。

我们现在就可以实现这个解决方案，但是让我们考虑一个现实世界中的复杂问题。

Noisy Sensors

完美的传感器是罕见的。也许，如果西蒙坐在门前抓挠自己，传感器就不会检测到门，或者如果他没有面对着走廊，传感器就会误读。所以当我找到门的时候，我不能用 1/3 作为概率。我必须将小于 1/3 的概率分配给每个门，并将小概率分配给每个空白的墙壁位置。类似的：

```
[.31, .31, .01, .01, .01, .01, .01, .01, .31, .01]
```

乍一看，这似乎是无法克服的。如果传感器有噪声，它会对每一份数据产生怀疑。如果我们总是不确定，我们怎么能得出结论呢?

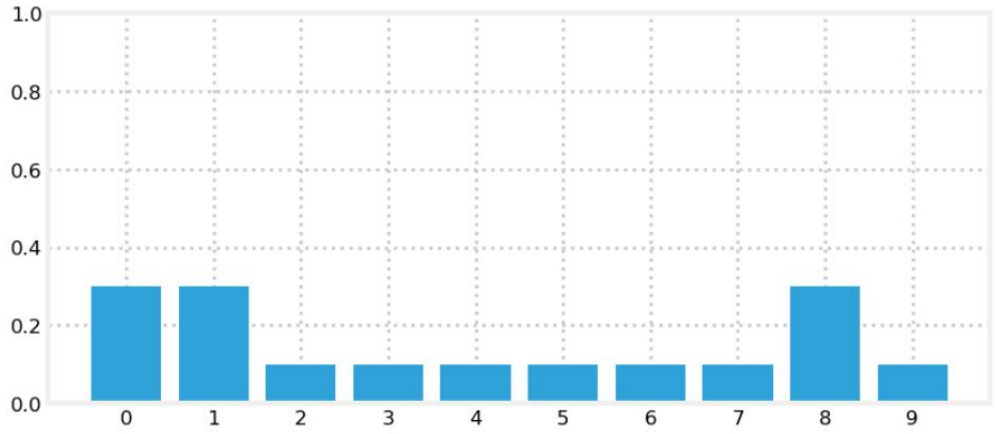
对于上述问题，答案是概率。我们已经习惯了给狗的位置分配一个概率信任;现在我们必须考虑由传感器噪声引起的额外不确定性。

假设我们读取了门的读数，并假设测试表明传感器的正确概率是错误的 3 倍。如果有一扇门，我们应该把概率分布乘以 3。如果我们这样做，结果将不再是一个概率分布，但我们一会儿会学习如何修正它。

让我们在 Python 代码中看看它。这里我用变量 z 来表示测量结果。Z 或 y 是文献中测量的习惯选择。作为一名程序员，我更喜欢有意义的变量名，但我希望您能够阅读文献和/或其他滤波代码，所以现在我将开始介绍这些缩写名称。

```
In [8]: def update_belief(hall, belief, z, correct_scale):
            for i, val in enumerate(hall):
                if val == z:
                    belief[i] *= correct_scale

        belief = np.array([0.1] * 10)
        reading = 1 # 1 is 'door'
        update_belief(hallway, belief, z=reading, correct_scale=3.)
        print('belief:', belief)
        print('sum =', sum(belief))
        plt.figure()
        book_plots.bar_plot(belief)
```

```
belief: [0.3 0.3 0.1 0.1 0.1 0.1 0.1 0.1 0.3 0.1]
sum = 1.6000000000000003
```



这不是一个概率分布，因为它的和不等于 1.0。但是代码做的大多是正确的事情——门被分配了一个数字(0.3)，比墙(0.1)高 3 倍。我们所需要做的就是将结果标准化，使概率正确地总和为 1.0。规范化是通过将每个元素除以列表中所有元素的和来实现的。这对 NumPy 来说很容易:

```
In [9]: belief / sum(belief)
Out[9]: array([0.188, 0.188, 0.062, 0.062, 0.062, 0.062, 0.062, 0.062, 0.188,
               0.062])
```

FilterPy 通过 normalize 函数实现了这一点:

```
from filterpy.discrete_bayes import normalize
normalize(belief)
```

说“正确的可能性是错误的三倍”有点奇怪。我们考虑的是概率，所以让我们指定传感

器正确的概率，并从中计算比例因子。方程是：

$$scale = \frac{prob_{correct}}{prob_{incorrect}} = \frac{prob_{correct}}{1 - prob_{correct}}$$

此外，for 循环也很麻烦。一般来说，你应该避免在 NumPy 代码中使用 for 循环。NumPy 是用 C 和 Fortran 实现的，因此如果避免 for 循环，结果运行速度通常比等效循环快 100 倍。

我们如何摆脱这个 for 循环?NumPy 允许使用布尔数组索引数组。使用逻辑运算符创建一个布尔数组。我们可以找到走廊里所有的门:

```
In  [10]: hallway == 1
Out[10]: array([ True,  True, False, False, False, False, False, False,  True,
                False])
```

当使用布尔数组作为另一个数组的索引时，它只返回索引为 True 的元素。因此，我们可以将 for 循环替换为：

```
belief[hall==z] *= scale
```

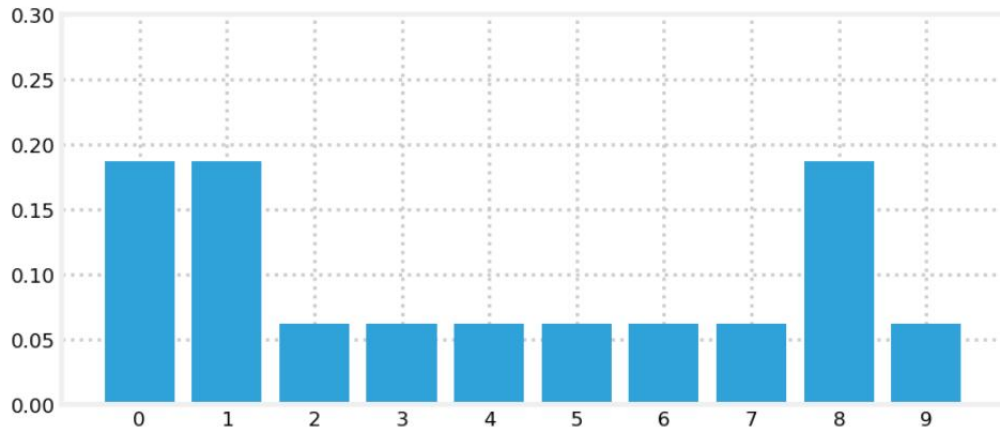
只有等于 z 的元素才会乘以比例。

教你 NumPy 超出了本书的范围。我将使用惯用的 NumPy 结构，并在第一次介绍它们时进行解释。如果您是 NumPy 的新手，有许多关于如何高效和惯用地使用 NumPy 的博客文章和视频。

这是我们的改进版本:

```
In  [12]: from filterpy.discrete_bayes import normalize

def scaled_update(hall, belief, z, z_prob):
    scale = z_prob / (1. - z_prob)
    belief[hall==z] *= scale
    normalize(belief)

belief = np.array([0.1] * 10)
scaled_update(hallway, belief, z=1, z_prob=.75)

print('sum =', sum(belief))
print('probability of door =', belief[0])
print('probability of wall =', belief[2])
book_plots.bar_plot(belief, ylim=(0, .3))
```

```
sum = 1.0
probability of door = 0.1875
probability of wall = 0.06249999999999999
```



我们可以从输出中看到，现在的总和是 1.0，门 vs 墙的概率仍然是原来的 3 倍。结果也符合我们的直觉，即门的概率必须小于 0.333，而墙的概率必须大于 0.0。最后，它应该符合我们的直觉，我们还没有得到任何信息，可以让我们区分任何给定的门或墙的位置，所以所

有门的位置应该有相同的值，墙的位置也应该如此。

这个结果被称为后验，即后验概率分布。所有这些都是指纳入测量信息后的概率分布(后验在这里是指“之后”)。回顾一下，先验是在包含测量信息之前的概率分布。

另一个术语是可能性。当我们计算信任[hall==z] *=scale 时，我们计算的是每个位置被给予测量的可能性。可能性不是概率分布，因为它的和不等于 1。

它们的组合给出了方程：

$$\text{posterior} = \frac{\text{likelihood} \times \text{prior}}{\text{normalization}}$$

当我们讨论滤波器的输出时，我们通常称执行预测后的状态为先验或预测，我们称更新后的状态为后验或估计状态。

学习和内化这些术语是非常重要的，因为大多数文献都广泛使用它们。

scaled_update()执行这个计算吗?是的，让我把它改写成这样:

In [13]:
```
def scaled_update(hall, belief, z, z_prob):
    scale = z_prob / (1. - z_prob)
    likelihood = np.ones(len(hall))
    likelihood[hall==z] *= scale
    return normalize(likelihood * belief)
```

这个函数不是完全通用的。它包含了关于走廊的知识，以及我们如何对其进行测量。我们总是努力写出一般的函数。在这里，我们将从函数中移除可能性的计算，并要求调用者自己计算可能性。

以下是该算法的完整实现:

```
def update(likelihood, prior):
    return normalize(likelihood * prior)
```

概率的计算因问题而异。例如，传感器可能不只是返回 1 或 0，而是返回一个介于 0 和 1 之间的浮点数，表示出现在门前的概率。它可能会使用计算机视觉，报告一个斑点形状，然后你可以概率地将其与门匹配。它可能使用声纳并返回一个距离读数。在每一种情况下，可能性的计算将是不同的。我们将在本书中看到许多这样的例子，并学习如何执行这些计算。

FilterPy 实现 Update。以下是前一个例子的完全一般形式:

In [14]:
```
from filterpy.discrete_bayes import update

def lh_hallway(hall, z, z_prob):
    """ compute likelihood that a measurement matches
    positions in the hallway."""

    try:
        scale = z_prob / (1. - z_prob)
    except ZeroDivisionError:
        scale = 1e8

    likelihood = np.ones(len(hall))
    likelihood[hall==z] *= scale
    return likelihood

belief = np.array([0.1] * 10)
likelihood = lh_hallway(hallway, z=1, z_prob=.75)
update(likelihood, belief)
```

Out[14]:
```
array([0.188, 0.188, 0.062, 0.062, 0.062, 0.062, 0.062, 0.062, 0.188,
       0.062])
```

Incorporating Movement

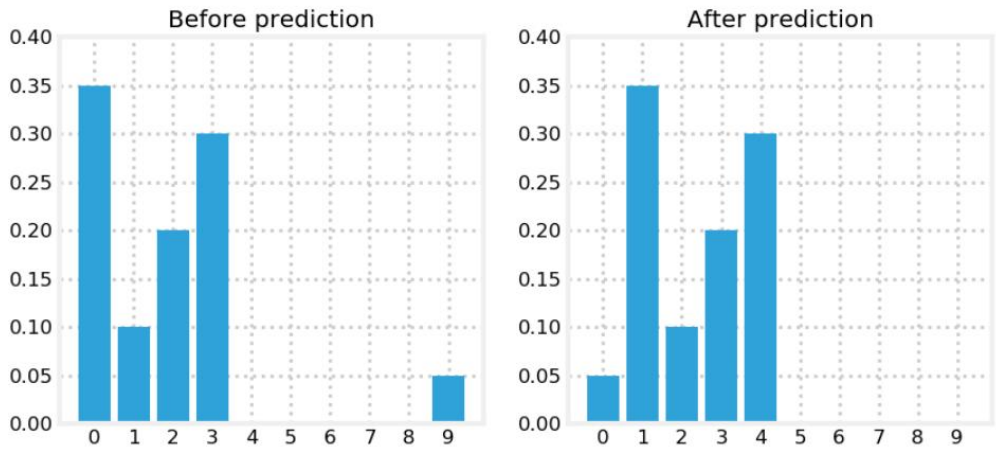
回想一下，当我们整合了一系列测量和移动更新时，我们是如何快速地找到一个精确的解决方案的。然而，这发生在一个完美传感器的虚构世界。也许我们可以用有噪声的传感器找到一个精确的解决方案?

不幸的是，答案是否定的。即使传感器读数与极其复杂的走廊地图完美匹配，我们也不能 100%确定狗狗处于特定的位置——毕竟，传感器读数有微小的可能是错误的!自然，在更典型的情况下，大多数传感器读数都是正确的，我们可能接近于 100%确定我们的答案，但从来没有 100%确定。这可能看起来很复杂，但是让我们继续编写数学程序。

首先让我们来处理一个简单的情况——假设运动传感器是完美的，它报告说狗向右移动了一个空格。我们如何改变信任数组?

我希望经过片刻的思考，我们可以清楚地看到，我们应该把所有的值向右移一个空格。如果我们之前认为西蒙在 3 号位置的概率是 50%，那么当他向右移动一个位置后我们应该相信他在 4 号位置的概率是 50%。走廊是圆形的，所以我们将使用模算法来执行移位。

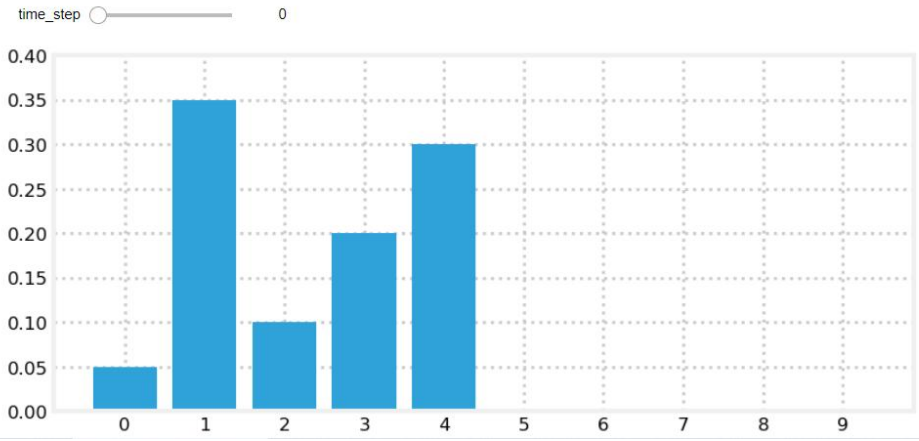
In [16]:
```
def perfect_predict(belief, move):
    """ move the position by `move` spaces, where positive is
    to the right, and negative is to the left
    """
    n = len(belief)
    result = np.zeros(n)
    for i in range(n):
        result[i] = belief[(i-move) % n]
    return result

belief = np.array([.35, .1, .2, .3, 0, 0, 0, 0, 0, .05])
plt.subplot(121)
book_plots.bar_plot(belief, title='Before prediction', ylim=(0, .4))

belief = perfect_predict(belief, 1)
plt.subplot(122)
book_plots.bar_plot(belief, title='After prediction', ylim=(0, .4))
```



我们可以看到，我们正确地将所有值右移了一个位置，并从数组的末尾返回到数组的开头。

下一个单元格将使其动画化，这样您就可以看到它的运行情况。使用滑块在时间上向前和向后移动。这模拟了西蒙在走廊里一圈一圈地走。它还没有纳入新的测量方法，所以概率分布不会改变形状，只会改变位置。

In [17]:
```
from ipywidgets import interact, IntSlider

belief = np.array([.35, .1, .2, .3, 0, 0, 0, 0, 0, .05])
perfect_beliefs = []

for _ in range(20):
    # Simon takes one step to the right
    belief = perfect_predict(belief, 1)
    perfect_beliefs.append(belief)

def simulate(time_step):
    book_plots.bar_plot(perfect_beliefs[time_step], ylim=(0, .4))
    plt.show()

interact(simulate, time_step=IntSlider(value=0, max=len(perfect_beliefs)-1));
```



Terminology

让我们暂停一下来复习一下术语。我在上一章介绍了这个术语，但是让我们花点时间来巩固您的知识。

这个系统是我们试图建模或滤波的。这里的系统就是我们的狗。状态是它当前的配置或值。在这一章中，状态是我们的狗的位置。我们很少知道实际的状态，所以我们说我们的滤波器产生了系统的估计状态。在实践中，这通常被称为状态，所以要注意理解上下文。

一个用测量进行预测和更新的周期称为状态或系统演化，简称时间演化[7]。另一个术语是系统传播。它指的是系统状态随时间的变化。对于过滤器，时间通常是一个离散的步骤，如 1 秒。对于我们的狗追踪器，系统状态是狗的位置，状态演化是经过一段离散时间后的位置。

我们使用流程模型对系统行为进行建模。在这里，我们的流程模型是狗在每个时间步移动一个或多个位置。这并不是一个特别准确的狗的行为模式。模型中的错误称为系统错误或过程错误。

预测就是我们的新先验。随着时间的推移，我们在不知道测量值的情况下做出了预测。

我们来看一个例子。狗的当前位置是 17 米。我们的 epoch 是 2 秒，狗以 15 米/秒的速度行进。我们预测他两秒钟后会在哪里?

清楚地，

$$\begin{aligned}\bar{x} &= 17 + (15 * 2) \\ &= 47\end{aligned}$$

我在变量上使用条形图表示它们是先验(预测)。我们可以这样写流程模型的方程:

$$\bar{x}_{k+1} = f_x(\bullet) + x_k$$

x_k 是当前的位置或状态。如果狗在 17 米，那么 $x_k = 17.$ 。

$f_x(\bullet)$ 是 x 的状态传播函数。它描述了 x_k 在一个时间步上的变化量。在我们的例子中，计算结果为 15x2，因此我们将其定义为:

$$f_x(v_x, t) = v_k t$$

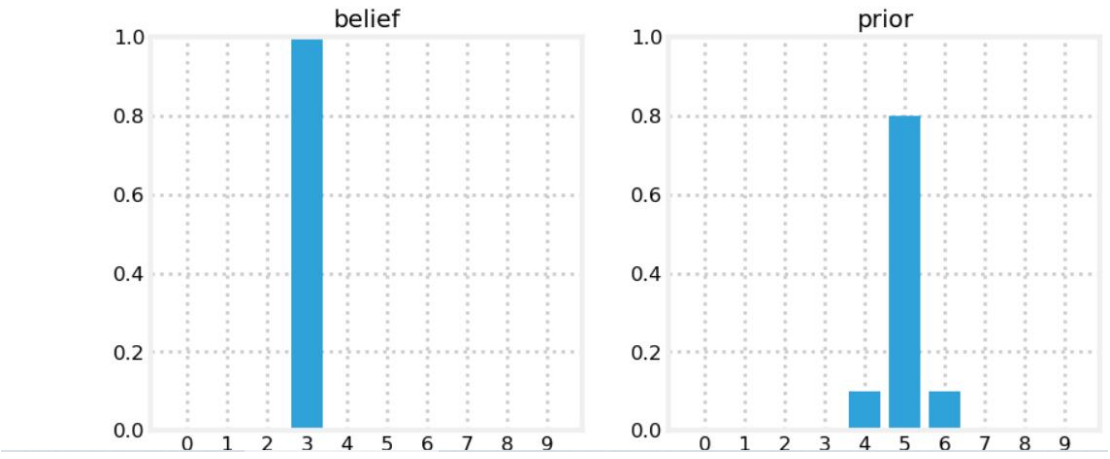
Adding Uncertainty to the Prediction

Perfect_predict()假设完美的测量，但所有传感器都有噪声。如果传感器报告说我们的狗移动了一个空间，但它实际上移动了两个空间，或者没有移动呢?这听起来似乎是一个难以克服的问题，但让我们对其进行建模，看看会发生什么。

假设传感器的运动测量有 80%的可能是正确的，10%的可能超过一个位置到右边，10%的可能低于一个位置到左边。也就是说，如果移动测量值是 4(意味着向右移动 4 个空格)，狗有 80%可能向右移动了 4 个空格，10%可能移动了 3 个空格，10%可能移动了 5 个空格。

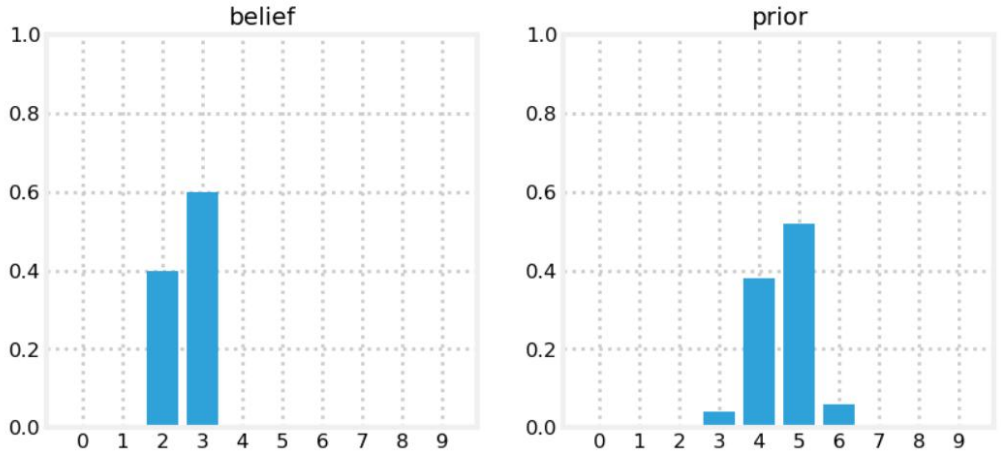
数组中的每个结果现在需要合并 3 种不同情况的概率。例如，考虑报告的移动 2。如果我们 100%确定狗从 3 号位置开始，那么它有 80%的机会在 5 号位置，有 10%的机会在 4 号或 6 号位置。让我们试着编写代码:

```
In [19]: def predict_move(belief, move, p_under, p_correct, p_over):
             n = len(belief)
             prior = np.zeros(n)
             for i in range(n):
                 prior[i] = (
                     belief[(i-move) % n]   * p_correct +
                     belief[(i-move-1) % n] * p_over +
                     belief[(i-move+1) % n] * p_under)
             return prior

         belief = [0., 0., 0., 1., 0., 0., 0., 0., 0., 0.]
         prior = predict_move(belief, 2, .1, .8, .1)
         book_plots.plot_belief_vs_prior(belief, prior)
```



它似乎工作正常。那么当我们的信任不是 100%确定的时候会发生什么呢?

```
In [22]: belief = [0, 0, .4, .6, 0, 0, 0, 0, 0, 0]
         prior = predict_move(belief, 2, .1, .8, .1)
         book_plots.plot_belief_vs_prior(belief, prior)
         prior

Out[22]: array([0.  , 0.  , 0.  , 0.04, 0.38, 0.52, 0.06, 0.  , 0.  , 0.  ])
```



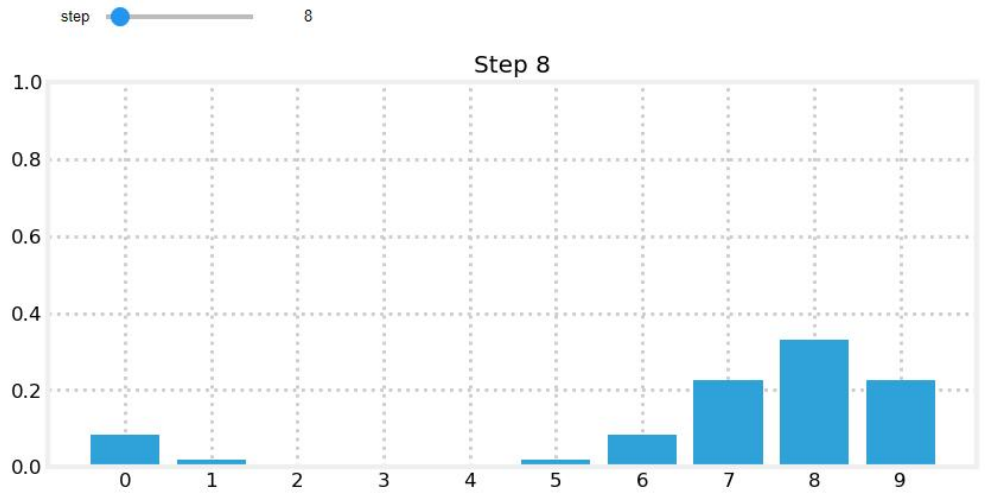
这里的结果更复杂，但你仍然可以心算出来。0.04 是由于 0.4 的信任差了 1 的可能性。0.38 是由于以下原因:我们移动 2 个位置(0.4 × 0.8)的概率为 80%，而我们不足(0.6 × 0.1)的概率为 10%。超调在这里没有作用，因为如果我们同时超调 0.4 和 0.6 就会超过这个位置。我强烈建议先做一些示例，直到所有这些都非常清楚为止，因为接下来的很多内容都取决于对这一步的理解。

如果你在执行更新后查看概率，你可能会感到沮丧。在上面的例子中，我们从两个位置的概率为 0.4 和 0.6 开始;执行更新之后，概率不仅会降低，而且会散布在地图上。

这不是一个巧合，也不是一个精心选择的例子的结果——它总是正确的预测。如果传感器是有噪声的，我们就会失去每一个预测的一些信息。假设我们要进行无限次的预测，结果会是什么?如果我们在每一步都丢失了信息，我们最终一定会失去任何信息，我们的概率将在整个信任数组中平均分布。让我们尝试 100 次迭代。情节是动画的;使用滑块更改步数。

In [23]:
```
belief = np.array([1.0, 0, 0, 0, 0, 0, 0, 0, 0, 0])
predict_beliefs = []

for i in range(100):
    belief = predict_move(belief, 1, .1, .8, .1)
    predict_beliefs.append(belief)

print('Final Belief:', belief)

# make interactive plot
def show_prior(step):
    book_plots.bar_plot(predict_beliefs[step-1])
    plt.title(f'Step {step}')
    plt.show()

interact(show_prior, step=IntSlider(value=1, max=len(predict_beliefs)));
```

```
Final Belief: [0.104 0.103 0.101 0.099 0.097 0.096 0.097 0.099 0.101 0.103]
```



In [24]:
```
print('Final Belief:', belief)
```

```
Final Belief: [0.104 0.103 0.101 0.099 0.097 0.096 0.097 0.099 0.101 0.103]
```

如果你在网上看这个，这里有一个输出的动画。

在本书的其余部分，我不会生成这些独立的动画。请参阅序言，以说明如何在网上免费运行这本书，或在您的计算机上安装 IPython。这将允许您运行所有单元格并查看动画。练习这些代码是非常重要的，而不是被动地阅读。

Generalizing with Convolution

我们假设运动误差最多为一个位置。但是误差有可能是 2 个，3 个，或更多的位置。作为程序员，我们总是希望将代码一般化，使其适用于所有情况。

这很容易用卷积来解决。卷积是用一个函数修改另一个函数。在我们的例子中，我们用传感器的误差函数来修改概率分布。predict_move()的实现是一个卷积，尽管我们没有这样称呼它。形式上，卷积被定义为：

$$(f * g)(t) = \int_0^t f(\tau)\, g(t-\tau)\, \mathrm{d}\tau$$

其中 $f * g$ 是 f 与 g 卷积的符号。它不是相乘的意思。

积分适用于连续函数，但我们用的是离散函数。我们用求和代替积分，用数组括号代替圆括号。

$$(f * g)[t] = \sum_{\tau=0}^{t} f[\tau]\, g[t-\tau]$$

比较显示 predict_move()正在计算这个方程—它计算一系列乘法的和。

Khan Academy[4]有一个关于卷积的很好的介绍，Wikipedia 也有一些关于卷积[5]的很棒的动画。但总的思路已经很清楚了。您将一个名为内核的数组滑动到另一个数组，将当前单元格的邻居与第二个数组的值相乘。在上面的例子中，我们使用 0.8 表示移动到正确位置的概率，0.1 表示欠冲，0.1 表示超冲。我们用数组[0.1,0.8,0.1]来做一个核。我们所需要做的就是编写一个循环，遍历数组中的每个元素，乘以内核，然后对结果求和。为了强调信任是一个概率分布，我把它命名为 pdf。

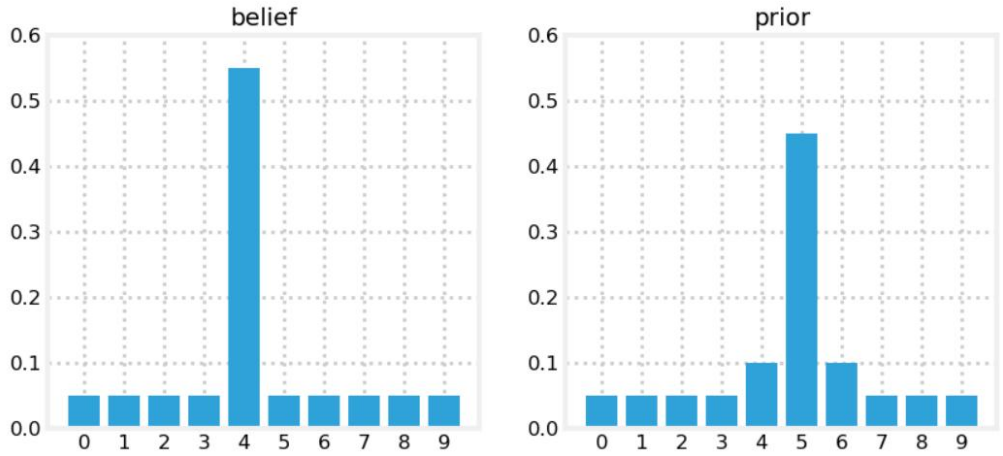
In [25]:
```
def predict_move_convolution(pdf, offset, kernel):
    N = len(pdf)
    kN = len(kernel)
    width = int((kN - 1) / 2)

    prior = np.zeros(N)
    for i in range(N):
        for k in range (kN):
            index = (i + (width-k) - offset) % N
            prior[i] += pdf[index] * kernel[k]
    return prior
```

这说明了算法，但它运行非常慢。SciPy 在 nimage 中提供了一个卷积例程 convolve()。过滤模块。我们需要在卷积前偏移 pdf;np.roll()。移动和预测算法可以用一行代码实现:

```
convolve(np.roll(pdf, offset), kernel, mode='wrap')
```

FilterPy 通过离散贝叶斯函数 predict()实现了这一点。

In [26]:
```
from filterpy.discrete_bayes import predict

belief = [.05, .05, .05, .05, .55, .05, .05, .05, .05, .05]
prior = predict(belief, offset=1, kernel=[.1, .8, .1])
book_plots.plot_belief_vs_prior(belief, prior, ylim=(0,0.6))
```



除了中间的元素，其他元素都没有变化。位置 4 和 6 的值应该是：

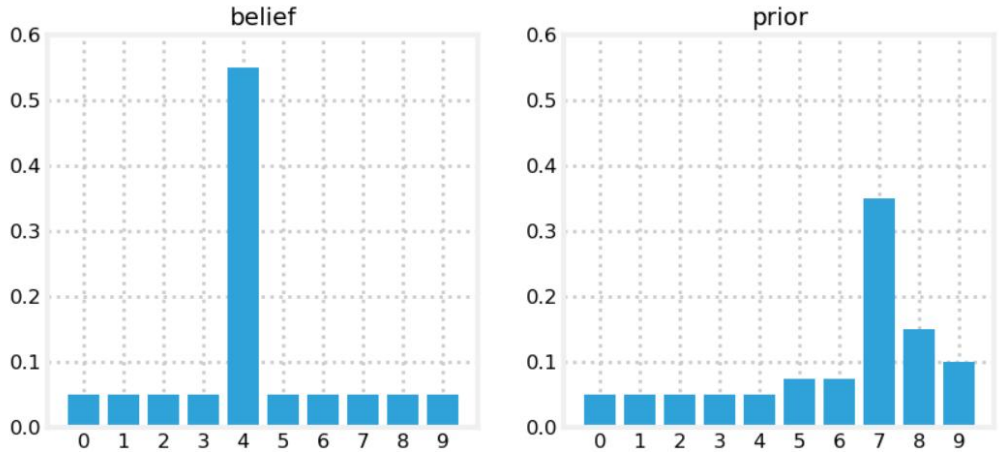
$$(0.1 \times 0.05) + (0.8 \times 0.05) + (0.1 \times 0.55) = 0.1$$

5 号位应：

$$(0.1 \times 0.05) + (0.8 \times 0.55) + (0.1 \times 0.05) = 0.45$$

让我们确保对于大于 1 的移动和非对称内核，它正确地移动了位置。

```
In [27]: prior = predict(belief, offset=3, kernel=[.05, .05, .6, .2, .1])
         book_plots.plot_belief_vs_prior(belief, prior, ylim=(0,0.6))
```



位置被正确地移动了 3 个位置，我们对超调和欠调的可能性给予了更多的权重，所以这看起来是正确的。确保你明白我们在做什么。我们对狗的移动方向进行预测，并对概率进行卷积以获得先验信息。如果我们不使用概率，我们可以使用我之前给出的这个方程:

$$\bar{x}_{k+1} = x_k + f_x(\bullet)$$

先验，我们对狗将会在哪里的预测，是狗移动的数量加上它当前的位置。狗在 10 米，它移动了 5 米，所以它现在在 15 米。这再简单不过了。但是我们用概率来建模，所以我们的方程是:

$$\bar{\mathbf{x}}_{k+1} = \mathbf{x}_k * f_{\mathbf{x}}(\bullet)$$

我们将当前的概率位置估计与我们认为狗移动了多少的概率估计进行卷积。概念是一样的，但计算方法略有不同。**x** 以粗体表示它是一个数字数组。

Integrating Measurements and Movement Updates

在预测过程中丢失信息的问题可能会使我们的系统看起来似乎会迅速退化到没有知识。然而，每一个预测之后都有一个更新，我们将测量纳入到估计中。这次更新提高了我们的知识。更新步骤的输出被输入到下一个预测中。这种预测降低了我们的确定性。这被传递到另一个更新，在那里确定性再次增加。

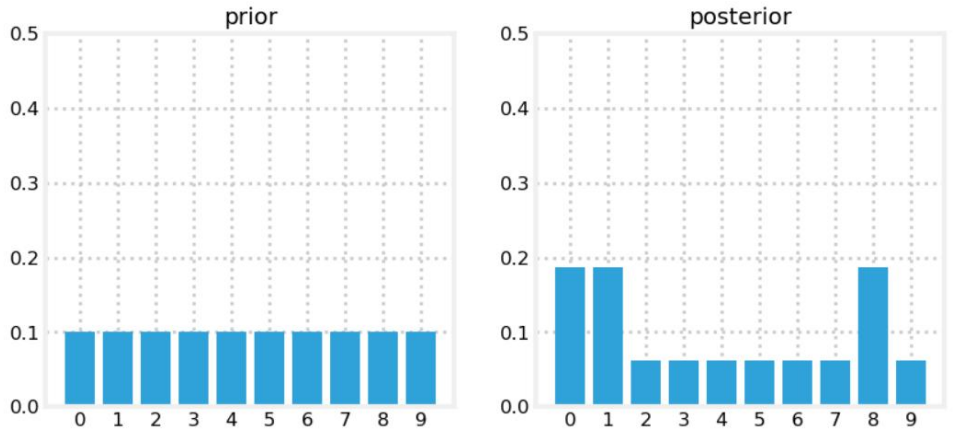
我们直观地考虑一下。考虑一个简单的例子——你在跟踪一只狗，而它却坐着不动。每次你预测他不会动。你的过滤器很快就能精确估计出他的位置。然后厨房的微波炉打开了，他就奔去了。你不知道这个，所以在下一个预测中你预测他在同一个位置。但测量结果却告诉我们一个不同的位置。当你把测量结果整合到一起时，你的信任就会被涂抹在通往厨房的走廊上。在每一个时期(周期)，你认为他坐着不动的想法会越来越少，你认为他正以惊人的速度朝厨房走来的想法也会越来越多。

这就是直觉告诉我们的。数学告诉我们什么?

我们已经安排了更新和预测步骤。所有我们需要做的是将一个结果输入到另一个，我们将实现一个狗跟踪器!!让我们看看它的表现。我们将输入测量值，就好像狗从位置 0 开始，

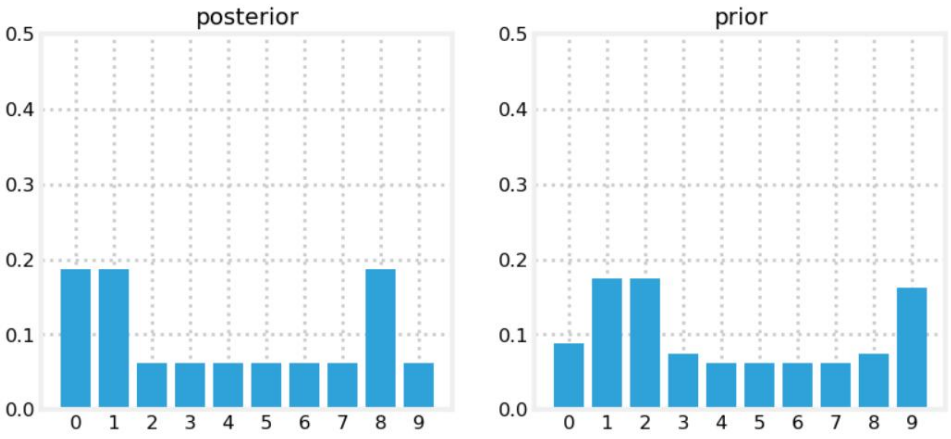
每个 epoch 向右移动一个位置。在真实世界的应用程序中，我们一开始不知道他的位置，将所有位置的概率相等。

```
In [29]: from filterpy.discrete_bayes import update

         hallway = np.array([1, 1, 0, 0, 0, 0, 0, 0, 1, 0])
         prior = np.array([.1] * 10)
         likelihood = lh_hallway(hallway, z=1, z_prob=.75)
         posterior = update(likelihood, prior)
         book_plots.plot_prior_vs_posterior(prior, posterior, ylim=(0,.5))
```



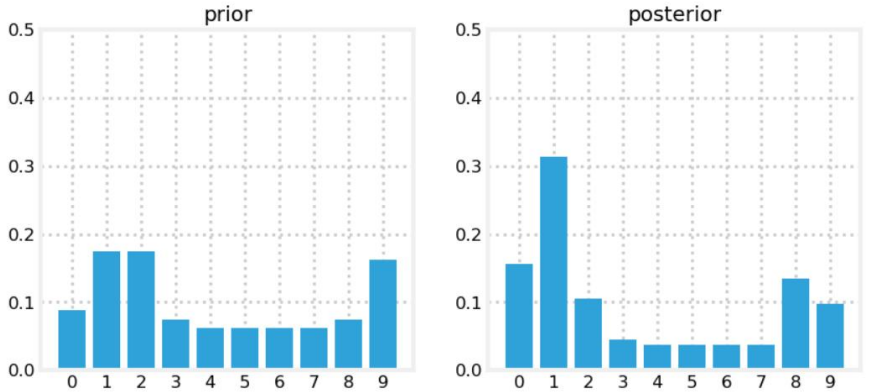
在第一次更新后，我们给每个门位置分配了一个高概率，给每个墙位置分配了一个低概率。

```
In [30]: kernel = (.1, .8, .1)
         prior = predict(posterior, 1, kernel)
         book_plots.plot_prior_vs_posterior(prior, posterior, True, ylim=(0,.5))
```



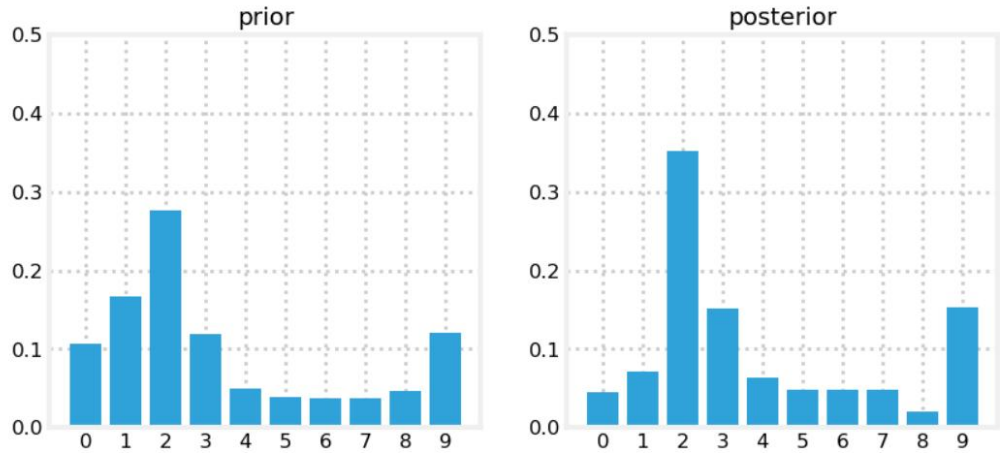
预测步骤将这些概率移到了右边，模糊了一些。现在让我们看看下一种感官会发生什么。

```
In [31]: likelihood = lh_hallway(hallway, z=1, z_prob=.75)
         posterior = update(likelihood, prior)
         book_plots.plot_prior_vs_posterior(prior, posterior, ylim=(0,.5))
```



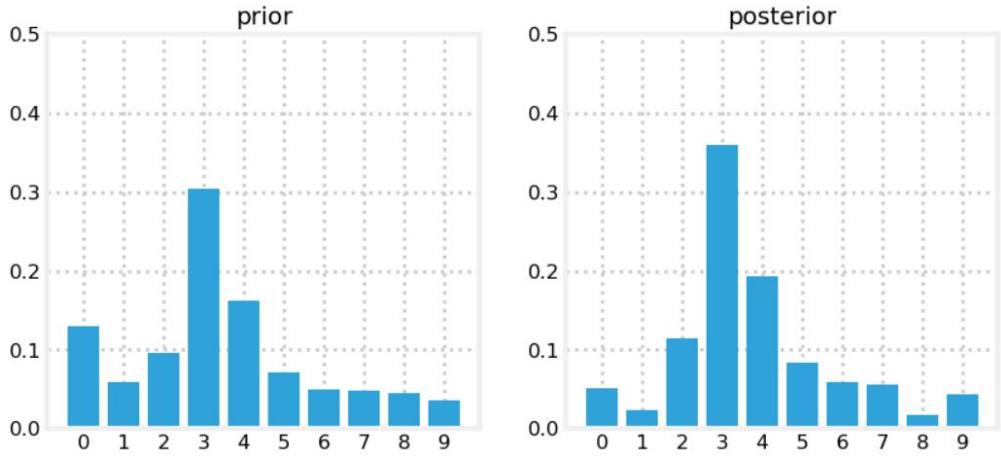
注意 1 号位置的高杆。这对应于(正确的)情况，即从位置 0 开始，感知一扇门，向右移动 1，然后感知另一扇门。没有任何其他立场能让这种观察结果成为可能。现在我们将添加一个更新，然后感知墙。

In [32]:
```
prior = predict(posterior, 1, kernel)
likelihood = lh_hallway(hallway, z=0, z_prob=.75)
posterior = update(likelihood, prior)
book_plots.plot_prior_vs_posterior(prior, posterior, ylim=(0,.5))
```



这是令人兴奋的!我们在第 2 位有一个非常突出的条形图，价值约为 35%。它的价值是该地块中其他条形物的两倍多，比我们上一个地块高出约 4%，当时最高的条形物约为 31%。再看一个循环。

In [33]:
```
prior = predict(posterior, 1, kernel)
likelihood = lh_hallway(hallway, z=0, z_prob=.75)
posterior = update(likelihood, prior)
book_plots.plot_prior_vs_posterior(prior, posterior, ylim=(0,.5))
```



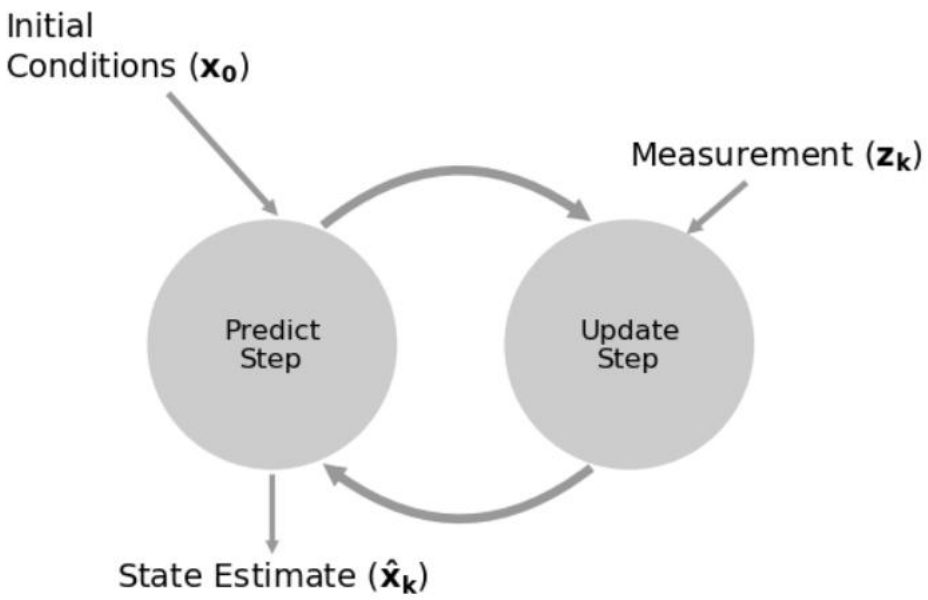
我忽略了一个重要的问题。之前我假设我们有一个运动传感器用于预测步长;然后，当谈到狗和微波炉时，我以为你不知道它突然开始跑了。我提到过，您认为狗在奔跑的想法会随着时间的推移而增加，但我没有为此提供任何代码。简而言之，如果我们不直接度量，我们如何检测和/或评估过程模型中的更改?

现在我想忽略这个问题。在后面的章节中，我们将学习这个估计背后的数学原理;目前，仅仅学习这个算法就已经是一个足够大的任务了。解决这个问题是非常重要的，但是我们还没有建立足够的所需的数学设备，所以在本章的剩余部分，我们将忽略这个问题，假设我们有一个感知运动的传感器。

The Discrete Bayes Algorithm

这个图表说明了算法:

```
In [31]: book_plots.predict_update_chart()
```



这个滤波器是 g-h 滤波器的一种形式。这里，我们使用错误的百分比隐式计算 g 和 h 参数。我们可以将离散贝叶斯算法表示为 g-h 滤波器，但这会模糊该滤波器的逻辑。

滤波方程为:

$$\bar{\mathbf{x}} = \mathbf{x} * f_{\mathbf{x}}(\bullet) \qquad \text{Predict Step}$$
$$\mathbf{x} = \|\mathcal{L} \cdot \bar{\mathbf{x}}\| \qquad \text{Update Step}$$

$\mathcal{L}$ 是似然函数的通常写法，所以我用它。‖ ‖表示采取规范。我们需要将概率与先验的乘积标准化，以确保 x 是一个和为 1 的概率分布。

我们可以用伪代码来表达这一点。

Initialization

初始化我们对状态的信任

Predict

根据系统的行为，预测下一个时间步的状态

调整信任以解释预测中的不确定性

Update

得到一个关于其准确性的测量和相关的信任

计算测量与每个状态匹配的可能性

用这种可能性更新状态信任

当我们讨论卡尔曼滤波时我们将使用完全相同的算法;只有计算的细节会有所不同。

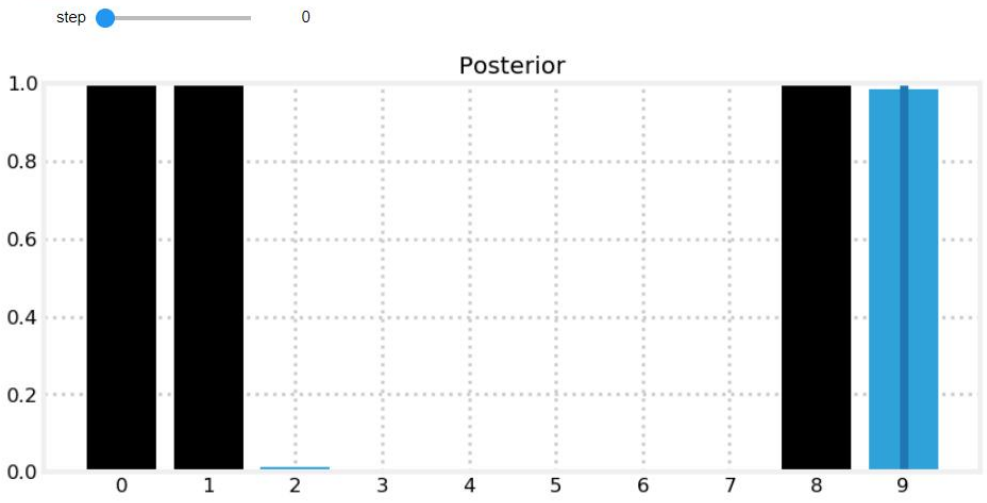
这种形式的算法有时被称为预测校正器。我们做出预测，然后修正它们。

让我们动画。首先，让我们编写一些函数来执行过滤并绘制任何步骤的结果。我用黑色标出了门口的位置。先验用橙色表示，后验用蓝色表示。我画了一条粗粗的竖线来表示西蒙的真实位置。这不是过滤器的输出-我们知道西蒙在哪里只是因为我们在模拟他的运动。

In [32]:
```
def discrete_bayes_sim(prior, kernel, measurements, z_prob, hallway):
    posterior = np.array([.1]*10)
    priors, posteriors = [], []
    for i, z in enumerate(measurements):
        prior = predict(posterior, 1, kernel)
        priors.append(prior)

        likelihood = lh_hallway(hallway, z, z_prob)
        posterior = update(likelihood, prior)
        posteriors.append(posterior)
    return priors, posteriors


def plot_posterior(hallway, posteriors, i):
    plt.title('Posterior')
    book_plots.bar_plot(hallway, c='k')
    book_plots.bar_plot(posteriors[i], ylim=(0, 1.0))
    plt.axvline(i % len(hallway), lw=5)
    plt.show()

def plot_prior(hallway, priors, i):
    plt.title('Prior')
    book_plots.bar_plot(hallway, c='k')
    book_plots.bar_plot(priors[i], ylim=(0, 1.0), c='#ff8015')
    plt.axvline(i % len(hallway), lw=5)
    plt.show()

def animate_discrete_bayes(hallway, priors, posteriors):
    def animate(step):
        step -= 1
        i = step // 2
        if step % 2 == 0:
            plot_prior(hallway, priors, i)
        else:
            plot_posterior(hallway, posteriors, i)

    return animate
```

让我们运行过滤器并使其动画化。

In [33]:
```
# change these numbers to alter the simulation
kernel = (.1, .8, .1)
z_prob = 1.0
hallway = np.array([1, 1, 0, 0, 0, 0, 0, 0, 1, 0])

# measurements with no noise
zs = [hallway[i % len(hallway)] for i in range(50)]

priors, posteriors = discrete_bayes_sim(prior, kernel, zs, z_prob, hallway)
interact(animate_discrete_bayes(hallway, priors, posteriors), step=IntSlider(value=1, max=len(zs)*2));
```



现在我们可以看到结果了。你可以看到先验如何改变位置，降低确定性，而后验保持不变，增加确定性，因为它纳入了测量的信息。我用直线 z_prob = 1.0 完美地实现了测量；我用直线 z_prob = 1.0 完美地实现了测量;在下一节中，我们将探讨不完美测量的影响。

另一件需要注意的事情是，当我们站在门前时，我们的估计会变得多么准确，而当我们站在走廊中间时，我们的估计又会变得多么不准确。这应该是直观的。这里只有几个门，所以当传感器告诉我们在一扇门前时，就会增强我们对自己位置的确定性。很长一段时间没有门会降低我们的确定性。

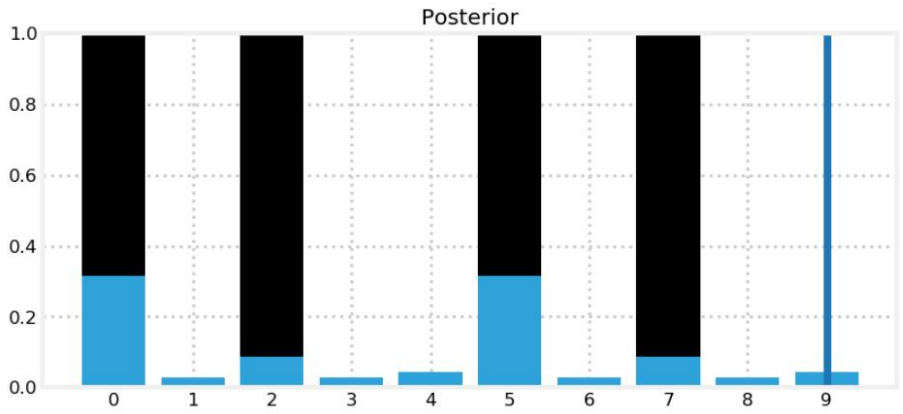
The Effect of Bad Sensor Data

您可能会对上面的结果产生怀疑，因为我总是将正确的传感器数据传递给函数。然而，我们声称这段代码实现了一个过滤器——它应该过滤掉不好的传感器测量。它能做到吗?

为了方便编程和可视化，我将改变走廊的布局，主要是交替门和走廊，并在 6 个正确的测量上运行算法:

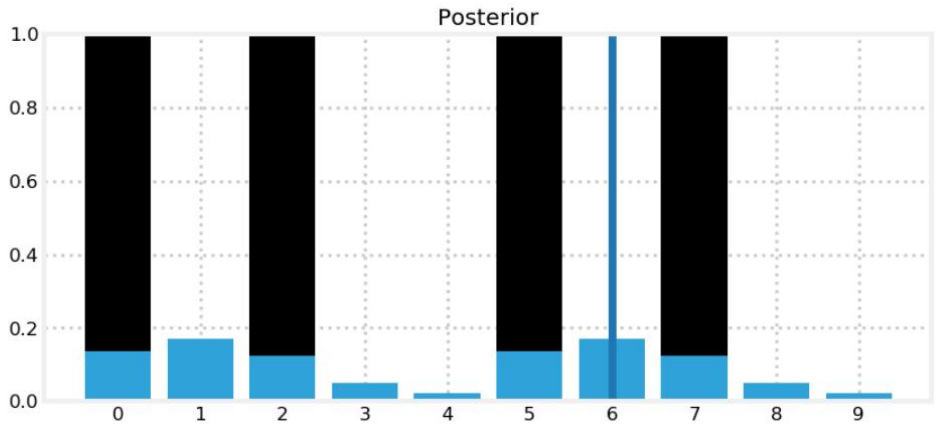
In [34]:
```
hallway = np.array([1, 0, 1, 0, 0]*2)
kernel = (.1, .8, .1)
prior = np.array([.1] * 10)
zs = [1, 0, 1, 0, 0, 1]
z_prob = 0.75
priors, posteriors = discrete_bayes_sim(prior, kernel, zs, z_prob, hallway)
interact(animate_discrete_bayes(hallway, priors, posteriors), step=IntSlider(value=12, max=len(zs)*2));
```

step 0



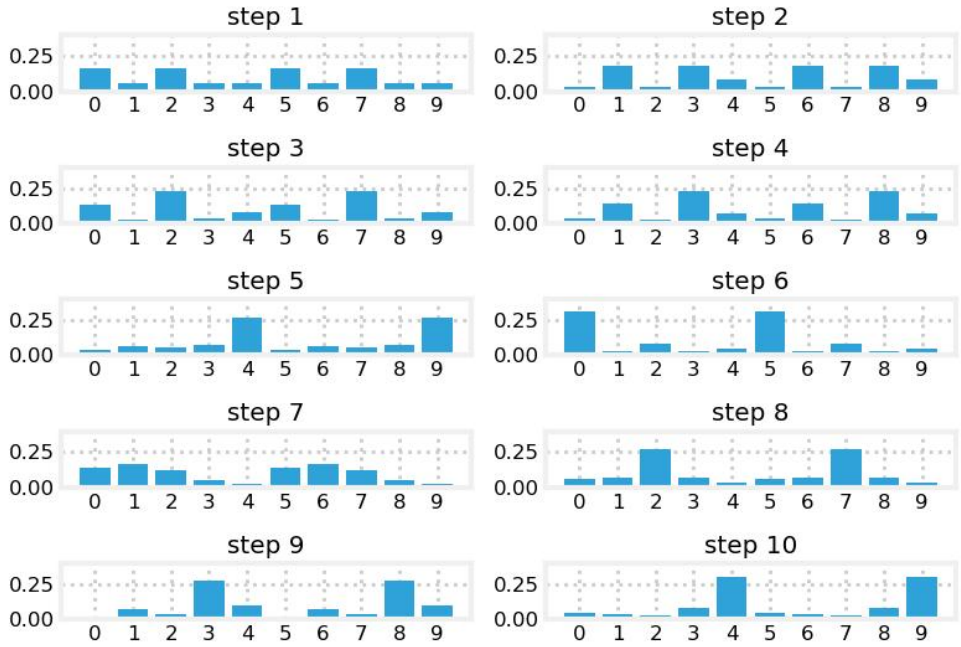
我们已经确定了从位置 0 或 5 开始的可能情况，因为我们看到了门和墙的序列:1,0,1,0,0。现在我注入了一个糟糕的测量方法。下一个测量值应该是 0，但我们得到的是 1:

In [35]:
```
measurements = [1, 0, 1, 0, 0, 1, 1]
priors, posteriors = discrete_bayes_sim(prior, kernel, measurements, z_prob, hallway);
plot_posterior(hallway, posteriors, 6)
```



一个糟糕的测量方法严重地侵蚀了我们的知识。现在让我们继续进行一系列正确的测量。

In [36]:
```
with figsize(y=5.5):
    measurements = [1, 0, 1, 0, 0, 1, 1, 1, 0, 0]
    for i, m in enumerate(measurements):
        likelihood = lh_hallway(hallway, z=m, z_prob=.75)
        posterior = update(likelihood, prior)
        prior = predict(posterior, 1, kernel)
        plt.subplot(5, 2, i+1)
        book_plots.bar_plot(posterior, ylim=(0, .4), title=f'step {i+1}')
    plt.tight_layout()
```



我们迅速过滤掉了糟糕的传感器读数，集中在我们的狗最可能的位置。

Drawbacks and Limitations

不要被我选择的例子的简单性所误导。这是一个健壮而完整的过滤器，您可以在实际的解决方案中使用该代码。如果你需要一个多模态的离散滤波器，这个滤波器可以工作。

尽管如此，这个过滤器并不经常使用，因为它有几个限制。绕开这些限制是本书后面章节的动机。

第一个问题是规模。我们的狗跟踪问题只使用了一个变量 pos，来表示狗的位置。最有趣的问题是要在一个大空间中跟踪几个东西。实际上，我们至少需要追踪狗狗的 (x, y) 坐标和他的速度 $(\dot{x}, \dot{y})$。我们还没有讨论多维的情况，但是我们使用多维网格来存储每个离散位置的概率，而不是数组。每个 update()和 predict()步骤都需要更新网格中的所有值，因此一个简单的四变量问题每个时间步骤需要 $O(n^4)$。现实的过滤器可能有 10 个或更多的变量要跟踪，导致过高的计算需求。

第二个问题是，过滤器是离散的，但我们生活在一个连续的世界。直方图要求您将过滤器的输出建模为一组离散点。一个 100 米的走廊需要 10000 个位置来模拟走廊到 1 厘米的

精度。因此，每次更新和预测操作都需要对 10,000 种不同的概率进行计算。随着维度的增加，情况会呈指数级恶化。一个 100x100m2 的庭院需要 1 亿个垃圾桶才能达到 1cm 的精度。

第三个问题是滤波器是多模态的。在上一个例子中，我们最终坚信狗在 4 号或 9 号位置。这并不总是一个问题。粒子滤波器，我们将在后面研究，是多模态的，并经常使用，因为这个性质。但想象一下，如果你车里的 GPS 告诉你，40%确定你在 D 街，30%确定你在柳树大道。

第四个问题是，它需要测量状态的变化。我们需要一个运动传感器来检测狗的运动程度。有一些方法可以解决这个问题，但它会使本章的阐述变得复杂，因此，鉴于前面提到的问题，我将不再进一步讨论它。

也就是说，如果我遇到这种技术可以处理的小问题，我就会选择使用它;实现、调试和理解所有优点都很简单。

Tracking and Control

我们一直在被动跟踪一个自主移动的物体。但是考虑这个非常相似的问题。我正在自动化一个仓库，并希望使用机器人来收集客户订单的所有项目。也许最简单的方法是让机器人在火车轨道上行走。我希望能把机器人送到一个目的地，让它到达那里。但火车轨道和机器人马达都不完美。车轮打滑和马达不完善意味着机器人不可能精确地到达你命令的位置。机器人不止一个，我们需要知道它们都在哪里，这样才不会导致它们坠毁。

所以我们添加了传感器。也许我们每隔几英尺就在轨道上安装一块磁铁，然后用霍尔传感器来计算有多少块磁铁通过了轨道。如果我们数到 10 块磁铁，那么机器人应该在第 10 块磁铁上。当然，有可能错过一块磁铁，或者数两次，所以我们必须考虑一定程度的误差。我们可以使用上一节中的代码来跟踪我们的机器人，因为磁铁计数非常类似于门口感应。

但我们还没有结束。我们学会了永远不要丢弃信息。如果你有信息，你应该用它来改进你的估计。我们遗漏了什么信息?我们知道在每个时刻，我们给机器人的轮子输入什么控制信息。例如，假设我们每一秒钟向机器人发送一个移动命令——向左移动一个单元，向右移动一个单元，或者原地不动。如果我发出“向左移动 1 单位”的命令，我预计在一秒钟后，机器人将在它现在所在的位置向左移动 1 单位。这是一个简化，因为我没有考虑到加速度，但我没有试图教控制理论。车轮和马达都不完美。机器人最终可能会到达 0.9 个单位，或者 1.2 个单位。

现在整个解决方案很清楚了。我们假设这只狗一直朝着它之前移动的方向移动。对我的狗来说，这是一个可疑的假设!机器人的可预测性要高得多。我们将输入发送给机器人的命令，而不是基于行为假设做出可疑的预测!换句话说，当我们调用 predict()时，我们将传入我们给机器人的指令运动，以及描述该运动可能性的内核。

Simulating the Train Behavior

我们需要模拟一列不完美的火车。当我们命令它移动时，它有时会犯一个小错误，它的传感器有时会返回错误的值。

In [37]:
```
class Train(object):

    def __init__(self, track_len, kernel=[1.], sensor_accuracy=.9):
        self.track_len = track_len
        self.pos = 0
        self.kernel = kernel
        self.sensor_accuracy = sensor_accuracy

    def move(self, distance=1):
        """ move in the specified direction
        with some small chance of error"""

        self.pos += distance
        # insert random movement error according to kernel
        r = random.random()
        s = 0
        offset = -(len(self.kernel) - 1) / 2
        for k in self.kernel:
            s += k
            if r <= s:
                break
            offset += 1
        self.pos = int((self.pos + offset) % self.track_len)
        return self.pos

    def sense(self):
        pos = self.pos
         # insert random sensor error
        if random.random() > self.sensor_accuracy:
            if random.random() > 0.5:
                pos += 1
            else:
                pos -= 1
        return pos
```

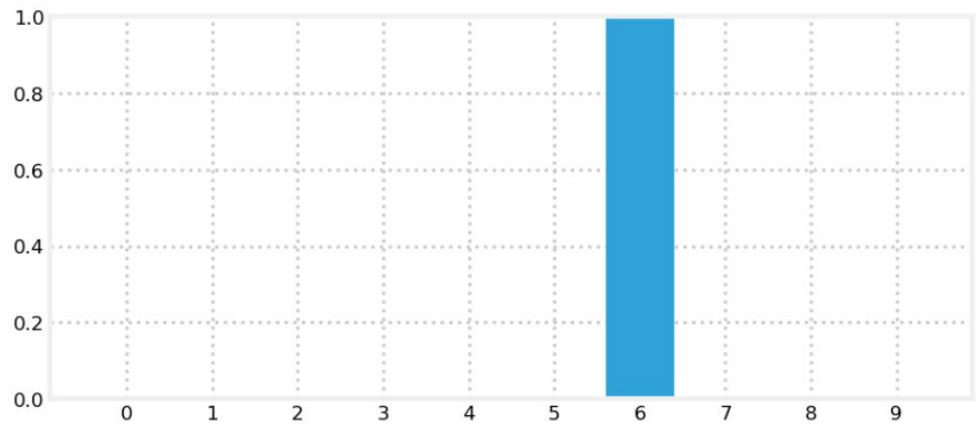
有了这些，我们就可以编写过滤器了。我们把它放在一个函数中这样我们就可以用不同的假设来运行它。我假设机器人总是从赛道的起点出发。轨道的长度是 10 个单位，但可以把它想象成一个长度为 10,000 个单位的轨道，磁体图案每 10 个单位重复一次。长度为 10 使绘图和检查更容易。

In [38]:
```
def train_filter(iterations, kernel, sensor_accuracy,
                 move_distance, do_print=True):
    track = np.array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
    prior = np.array([.9] + [0.01]*9)
    posterior = prior[:]
    normalize(prior)

    robot = Train(len(track), kernel, sensor_accuracy)
    for i in range(iterations):
        # move the robot and
        robot.move(distance=move_distance)

        # peform prediction
        prior = predict(posterior, move_distance, kernel)

        #  and update the filter
        m = robot.sense()
        likelihood = lh_hallway(track, m, sensor_accuracy)
        posterior = update(likelihood, prior)
        index = np.argmax(posterior)

        if do_print:
            print(f'time {i}: pos {robot.pos}, sensed {m}, at position {track[robot.pos]}')
            conf = posterior[index] * 100
            print(f'        estimated position is {index} with confidence {conf:.4f}%:')

    book_plots.bar_plot(posterior)
    if do_print:
        print()
        print('final position is', robot.pos)
        index = np.argmax(posterior)
        conf = posterior[index]*100
        print(f'Estimated position is {index} with confidence {conf:.4f}')
```

阅读代码并确保你理解它。现在让我们做一个没有传感器或移动错误的运行。如果代码是正确的，它应该能够毫无错误地定位机器人。输出读起来有点乏味，但是如果您完全不确定更新/预测周期是如何工作的，请确保仔细阅读它，以巩固您的理解。

In [39]:
```
import random

random.seed(3)
np.set_printoptions(precision=2, suppress=True, linewidth=60)
train_filter(4, kernel=[1.], sensor_accuracy=.999,
             move_distance=4, do_print=True)
```

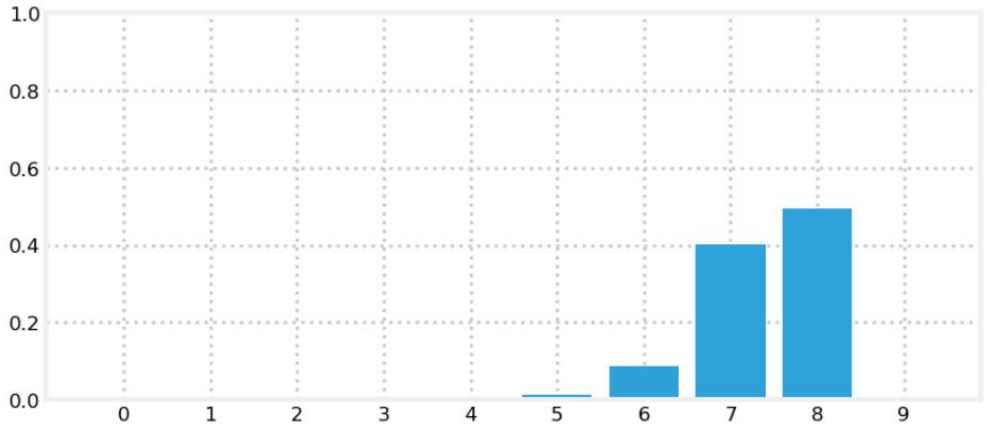
```
time 0: pos 4, sensed 4, at position 4
        estimated position is 4 with confidence 99.9900%:
time 1: pos 8, sensed 8, at position 8
        estimated position is 8 with confidence 100.0000%:
time 2: pos 2, sensed 2, at position 2
        estimated position is 2 with confidence 100.0000%:
time 3: pos 6, sensed 6, at position 6
        estimated position is 6 with confidence 100.0000%:

final position is 6
Estimated position is 6 with confidence 100.0000
```



我们可以看到代码能够完美地跟踪机器人，所以我们应该有理由相信代码是有效的。现在让我们看看它是如何处理一些错误的。

In [40]:
```
random.seed(5)
train_filter(4, kernel=[.1, .8, .1], sensor_accuracy=.9,
             move_distance=4, do_print=True)
```

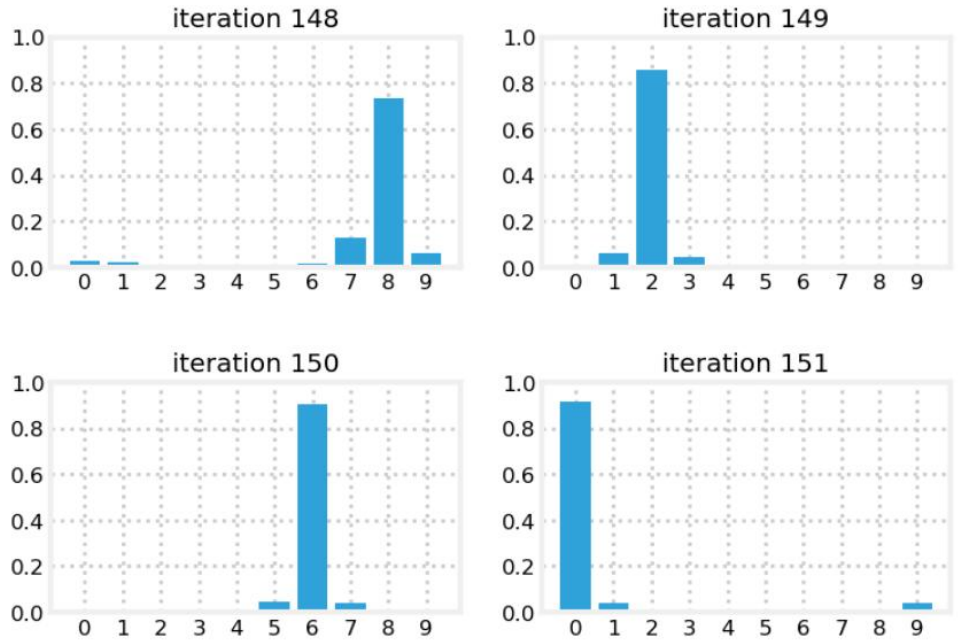
```
time 0: pos 4, sensed 4, at position 4
        estimated position is 4 with confidence 96.0390%:
time 1: pos 8, sensed 9, at position 8
        estimated position is 9 with confidence 52.1180%:
time 2: pos 3, sensed 3, at position 3
        estimated position is 3 with confidence 88.3993%:
time 3: pos 7, sensed 8, at position 7
        estimated position is 8 with confidence 49.3174%:

final position is 7
Estimated position is 8 with confidence 49.3174
```



时间 1 有一个感应错误，但我们仍然对我们的位置很有信心。

现在让我们运行一个非常长的模拟，看看过滤器如何响应错误。

In [41]:
```
with figsize(y=5.5):
    for i in range (4):
        random.seed(3)
        plt.subplot(221+i)
        train_filter(148+i, kernel=[.1, .8, .1],
                     sensor_accuracy=.8,
                     move_distance=4, do_print=False)
        plt.title (f'iteration {148 + i}')
```



我们可以看到，随着置信度的下降，迭代 149 出现了一个问题。但在几次迭代中，滤波器能够自我修正，并恢复对估计位置的信心。

Bayes Theorem and the Total Probability Theorem

在本章中，我们仅仅是通过对我们每时每刻所掌握的信息进行推理来发展数学的。在这个过程中，我们发现了贝叶斯定理和全概率定理。

贝叶斯定理告诉我们如何计算给定先前信息的事件发生的概率。

我们使用这样的概率计算来实现 update()函数:

$$\text{posterior} = \frac{\text{likelihood} \times \text{prior}}{\text{normalization factor}}$$

我们还没有发展出讨论贝叶斯的数学，但这就是贝叶斯定理。本书中的每个滤波器都是贝叶斯定理的表达。在下一章中，我们将发展数学，但在很多方面，它掩盖了这个方程所表达的简单思想:

$$updated\ knowledge = \|likelihood\ of\ new\ knowledge \times prior\ knowledge\|$$

其中，$\|\cdot\|$表示术语的规范化。

我们得出这个结论的原因很简单:一只狗在走廊上走。然而，正如我们将看到的，同样的方程适用于所有的过滤问题。我们将在以后的每一章中使用这个方程。

同样，predict()步骤计算多个可能事件的总概率。这在统计学中被称为全概率定理，我们在开发了一些辅助数学之后，也将在下一章中讨论这个问题。

现在我需要你们理解贝叶斯定理是一个公式，它将新的信息整合到现有的信息中。

Summary

代码非常短，但结果令人印象深刻!我们实现了贝叶斯过滤器的一种形式。我们已经学会了如何从无信息开始，从有噪声的传感器中获取信息。尽管本章中的传感器噪声很大(例如，大多数传感器的准确率都超过 80%)，我们还是能迅速集中到狗狗最可能出现的位置上。

我们已经知道，预测步骤总是会降低我们的知识水平，但加入另一种测量方法，即使其中可能有噪声，也会提高我们的知识水平，使我们能够集中于最可能的结果。

这本书主要是关于卡尔曼滤波器的。它使用的数学是不同的，但逻辑与本章使用的完全相同。它使用贝叶斯推理从测量和过程模型的组合形成估计。

如果你能理解这一章，你就能理解和实现卡尔曼滤波器。这一点我再强调也不为过。如果有什么不清楚的地方，请返回并重新阅读本章并摆弄代码。本书的其余部分将以我们在这里使用的算法为基础。如果你不明白为什么这个滤波会起作用，你就很难成功处理其他的材料。然而，如果你掌握了基本的洞察力——当我们测量时乘以概率，当我们更新时平移概率，就会得到一个收敛的解决方案——那么在学习了一些数学之后，你就可以实现卡尔曼滤波了。