We can call `tf.zeros()` and `tf.ones()` to create and display tensors of various sizes (Example 2-4).

Example 2-4. Evaluate and display tensors

```
>>> a = tf.zeros((2, 3))
>>> a.eval()
array([[ 0.,  0.,  0.],
       [ 0.,  0.,  0.]], dtype=float32)
>>> b = tf.ones((2,2,2))
>>> b.eval()
array([[[ 1.,  1.],
        [ 1.,  1.]],

       [[ 1.,  1.],
        [ 1.,  1.]]], dtype=float32)
```

What if we'd like a tensor filled with some quantity besides 0/1? The `tf.fill()` method provides a nice shortcut for doing so (Example 2-5).

Example 2-5. Filling tensors with arbitrary values

```
>>> b = tf.fill((2, 2), value=5.)
>>> b.eval()
array([[ 5.,  5.],
       [ 5.,  5.]], dtype=float32)
```

`tf.constant` is another function, similar to `tf.fill`, which allows for construction of tensors that shouldn't change during the program execution (Example 2-6).

Example 2-6. Creating constant tensors

```
>>> a = tf.constant(3)
>>> a.eval()
3
```

Sampling Random Tensors

Although working with constant tensors is convenient for testing ideas, it's much more common to initialize tensors with random values. The most common way to do this is to sample each entry in the tensor from a random distribution. `tf.random_normal` allows for each entry in a tensor of specified shape to be sampled from a Normal distribution of specified mean and standard deviation (Example 2-7).

Symmetry Breaking

Many machine learning algorithms learn by performing updates to a set of tensors that hold weights. These update equations usually satisfy the property that weights initialized at the same value will continue to evolve together. Thus, if the initial set of tensors is initialized to a constant value, the model won't be capable of learning much. Fixing this situation requires *symmetry breaking*. The easiest way of breaking symmetry is to sample each entry in a tensor randomly.

Example 2-7. Sampling a tensor with random Normal entries

```
>>> a = tf.random_normal((2, 2), mean=0, stddev=1)
>>> a.eval()
array([[-0.73437649, -0.77678096],
       [ 0.51697761,  1.15063596]], dtype=float32)
```

One thing to note is that machine learning systems often make use of very large tensors that often have tens of millions of parameters. When we sample tens of millions of random values from the Normal distribution, it becomes almost certain that some sampled values will be far from the mean. Such large samples can lead to numerical instability, so it's common to sample using `tf.truncated_normal()` instead of `tf.random_normal()`. This function behaves the same as `tf.random_normal()` in terms of API, but drops and resamples all values more than two standard deviations from the mean.

`tf.random_uniform()` behaves like `tf.random_normal()` except for the fact that random values are sampled from the Uniform distribution over a specified range (Example 2-8).

Example 2-8. Sampling a tensor with uniformly random entries

```
>>> a = tf.random_uniform((2, 2), minval=-2, maxval=2)
>>> a.eval()
array([[-1.90391684,  1.4179163 ],
       [ 0.67762709,  1.07282352]], dtype=float32)
```

Tensor Addition and Scaling

TensorFlow makes use of Python's operator overloading to make basic tensor arithmetic straightforward with standard Python operators (Example 2-9).