Creating Arrays from Python Lists

First, we can use `np.array` to create arrays from Python lists:

```
In[8]: # integer array:
       np.array([1, 4, 2, 5, 3])

Out[8]: array([1, 4, 2, 5, 3])
```

Remember that unlike Python lists, NumPy is constrained to arrays that all contain the same type. If types do not match, NumPy will upcast if possible (here, integers are upcast to floating point):

```
In[9]: np.array([3.14, 4, 2, 3])

Out[9]: array([ 3.14,  4.  ,  2.  ,  3.  ])
```

If we want to explicitly set the data type of the resulting array, we can use the `dtype` keyword:

```
In[10]: np.array([1, 2, 3, 4], dtype='float32')

Out[10]: array([ 1.,  2.,  3.,  4.], dtype=float32)
```

Finally, unlike Python lists, NumPy arrays can explicitly be multidimensional; here's one way of initializing a multidimensional array using a list of lists:

```
In[11]: # nested lists result in multidimensional arrays
        np.array([range(i, i + 3) for i in [2, 4, 6]])

Out[11]: array([[2, 3, 4],
                [4, 5, 6],
                [6, 7, 8]])
```

The inner lists are treated as rows of the resulting two-dimensional array.

Creating Arrays from Scratch

Especially for larger arrays, it is more efficient to create arrays from scratch using routines built into NumPy. Here are several examples:

```
In[12]: # Create a length-10 integer array filled with zeros
        np.zeros(10, dtype=int)

Out[12]: array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0])

In[13]: # Create a 3x5 floating-point array filled with 1s
        np.ones((3, 5), dtype=float)

Out[13]: array([[ 1.,  1.,  1.,  1.,  1.],
                [ 1.,  1.,  1.,  1.,  1.],
                [ 1.,  1.,  1.,  1.,  1.]])

In[14]: # Create a 3x5 array filled with 3.14
        np.full((3, 5), 3.14)
```