

## A Python List Is More Than Just a List

Let's consider now what happens when we use a Python data structure that holds many Python objects. The standard mutable multielement container in Python is the list. We can create a list of integers as follows:

```
In[1]: L = list(range(10))
      L
Out[1]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
In[2]: type(L[0])
Out[2]: int
```

Or, similarly, a list of strings:

```
In[3]: L2 = [str(c) for c in L]
      L2
Out[3]: ['0', '1', '2', '3', '4', '5', '6', '7', '8', '9']
In[4]: type(L2[0])
Out[4]: str
```

Because of Python's dynamic typing, we can even create heterogeneous lists:

```
In[5]: L3 = [True, "2", 3.0, 4]
      [type(item) for item in L3]
Out[5]: [bool, str, float, int]
```

But this flexibility comes at a cost: to allow these flexible types, each item in the list must contain its own type info, reference count, and other information—that is, each item is a complete Python object. In the special case that all variables are of the same type, much of this information is redundant: it can be much more efficient to store data in a fixed-type array. The difference between a dynamic-type list and a fixed-type (NumPy-style) array is illustrated in [Figure 2-2](#).

At the implementation level, the array essentially contains a single pointer to one contiguous block of data. The Python list, on the other hand, contains a pointer to a block of pointers, each of which in turn points to a full Python object like the Python integer we saw earlier. Again, the advantage of the list is flexibility: because each list element is a full structure containing both data and type information, the list can be filled with data of any desired type. Fixed-type NumPy-style arrays lack this flexibility, but are much more efficient for storing and manipulating data.

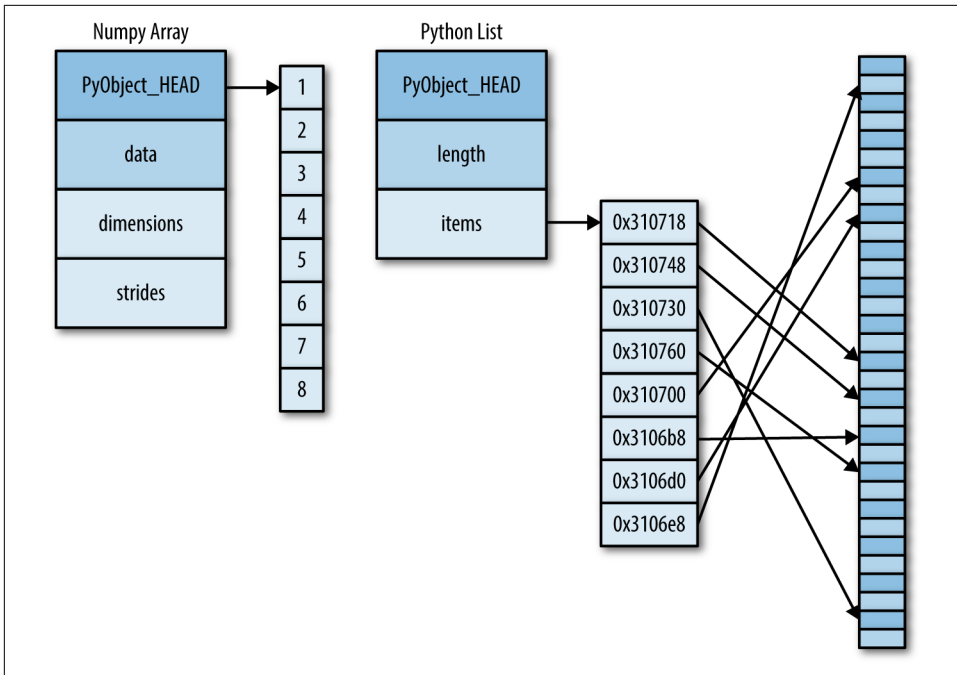


Figure 2-2. The difference between C and Python lists

## Fixed-Type Arrays in Python

Python offers several different options for storing data in efficient, fixed-type data buffers. The built-in `array` module (available since Python 3.3) can be used to create dense arrays of a uniform type:

```
In[6]: import array
      L = list(range(10))
      A = array.array('i', L)
      A

Out[6]: array('i', [0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

Here `'i'` is a type code indicating the contents are integers.

Much more useful, however, is the `ndarray` object of the NumPy package. While Python's `array` object provides efficient storage of array-based data, NumPy adds to this efficient *operations* on that data. We will explore these operations in later sections; here we'll demonstrate several ways of creating a NumPy array.

We'll start with the standard NumPy import, under the alias `np`:

```
In[7]: import numpy as np
```