

3.3: Arrays

We will often have to deal with collections of several numbers, which requires organizing them into **data structures**. One of the data structures that we will use most frequently is the **array**, which is a fixed-size linear sequence of numbers. We have already discussed the basic usage of Scipy arrays in the previous article.

The memory layout of an array is shown schematically in Fig. 3.3.1. It consists of two separate regions of memory:

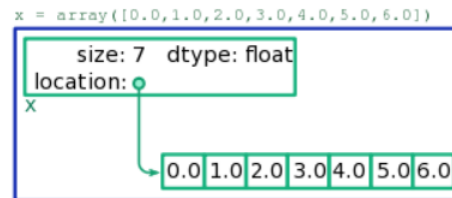


Figure 3.3.1: Schematic of how an array is laid out in memory. The book-keeping block (upper left box) records the array size, the address of the storage blocks (indicated by an arrow), etc. The storage blocks (lower right boxes) contain the array contents, in sequential order.

1. One region, which we call the **book-keeping block**, stores summary information about the array, including (i) the total number of elements, (ii) the memory address where the array contents are stored (specifically, the address of element 0), and (iii) the type of numbers stored in the array. The first two pieces of information are recorded in the form of integers, while the last piece is recorded in some other format that we don't need to worry about (it's managed by Python).
2. The second region, which we call the **data block**, stores the actual contents of the array, laid out sequentially. For example, for an array containing seven 64-bit integers, this block will consist of $7 \times 64 = 448$ bits of memory, storing the integers one after the other.

The book-keeping block and the data block aren't necessarily kept next to each other in memory. When a piece of Python code acts upon an array `x`, the information in the array's book-keeping block is used to locate the data block, and then access/alter its data as necessary.

3.3.1 Basic Array Operations

Let's take detailed look at what happens when we read or write an individual element of an array, say `x[2]`: the third element (index 2) stored in the array `x`.

From the array name, `x`, Python knows the address of the relevant book-keeping block (this is handled internally by Python, and takes negligible time). The book-keeping block records the address of element 0, i.e. the start of the data block. Because we want index 2 of the array, the processor jumps to the memory address that is 2 blocks past the recorded address. Since the data block is laid out sequentially, that is precisely the address where the number `x[2]` is stored. This number can now be read or overwritten, as desired by the Python code.

Under this scheme, the reading/writing of individual array elements is independent of the size of the array. Accessing an element in a size-1 array takes the same time as accessing an element in a size-100000 array. This is because the memory is random-access—the processor can jump to any address in memory once you tell it where to go. The memory layout of an array is designed so that one can always work out the relevant address in a single step.

We describe the speed of this operation using **big-O notation**. If N is the size of the array, reading/writing individual array elements is said to take $O(1)$ time, or "order-1 time" (i.e., independent of N). By contrast, a statement like

```
x.fill(3.3)
```

takes $O(N)$ time, i.e. time proportional to the array size N . That's because the `fill` method assign values to each of the N elements of the array. Similarly, the statement

```
x += 1.0
```

takes $O(N)$ time. This `+=` operation adds `1.0` to each of the elements of the array, which requires N arithmetic operations.

3.3.2 Array Data Type

We have noted that the book-keeping block of each array records the type of number, or **data type**, kept in the storage blocks. Thus, each individual array is able to store only one type of number. When you create an array with the `array` function, Scipy infers the data type based on the specified array contents. For example, if the input contains only integers, an integer array is created; if you then try to store a floating-point number, it will be rounded down to an integer:

```
>>> a = array([1,2,3,4])
>>> a[1] = 3.14159
>>> a
array([1, 3, 3, 4])
```

In the above situation, if our intention was to create an array of floating point numbers, that can be done by giving the `array` function an input containing at least one floating-point number. For example,

```
>>> a = array([1,2,3,4.])
>>> a[1] = 3.14159
>>> a
array([ 1.      ,  3.14159,  3.      ,  4.      ])
```

Alternatively, the `array` function accepts a parameter named `dtype`, which can be used to specify the data type directly:

```
>>> a = array([1,2,3,4], dtype=float)
>>> a[1] = 3.14159
>>> a
array([ 1.      ,  3.14159,  3.      ,  4.      ])
```

The `dtype` parameter accepts several possible values, but most of time you will choose one of these three:

- `float`
- `complex`
- `integer`

The common functions for creating new arrays, `zeros`, `ones`, and `linspace`, create arrays with the `float` data type by default. They also accept `dtype` parameters, in case you want a different data type. For example:

```
>>> a = zeros(4, dtype=complex)
>>> a[1] = 2.5+1j
>>> a
array([ 0.0+0.j,  2.5+1.j,  0.0+0.j,  0.0+0.j])
```

3.3.3 Vectorization

We have previously discussed the code `x += 1.0`, which adds `1.0` to every element on the array `x`. It has runtime $O(N)$, where N is the array length. We could also have done the same thing by looping over the array, as follows:

```
for n in range(len(x)):
    x[n] += 1.0
```

This, too, has $O(N)$ runtime. But it is not a good way to do the job, for two reasons. Firstly, it's obviously much more cumbersome to write. Secondly, and more importantly, it is much more inefficient, because it involves more "high-level" Python operations. To

run this code, Python has to create an index variable n , increment that index variable N times, and increment $x[n]$ for each separate value of n .

By contrast, when you write $x += 1.0$, Python uses "low-level" code to increment each element in the array, which does not require introducing and managing any "high-level" Python objects. The practice of using array operations, instead of performing explicit loops over an array, is called **vectorization**. You should always strive to vectorize your code; it is generally good programming practice, and leads to extreme performance gains for large array sizes.

Vectorization does not change the runtime scaling of the operation. The vectorized code $x += 1.0$, and the explicit loop, both run in $O(N)$ time. What changes is the *coefficient* of the scaling: the runtime has the form T , and the value of the coefficient a is much smaller for vectorized code.

Here is another example of vectorization. Suppose we have a variable y whose value is a number, and an array x containing a collection of numbers; we want to find the element of x closest to y . Here is non-vectorized code for doing this:

```
idx, distance = 0, abs(x[0] - y)
for n in range(1, len(x)):
    new_dist = abs(x[n] - y)
    if new_dist < distance:
        idx, distance = n, new_dist

z = x[idx]
```

The vectorized approach would simply make use of the `argmin` function:

```
idx = argmin(abs(x - y))
z = x[idx]
```

The way this works is to create a new array, whose values are the distances between each element of x and the target number y ; then, `argmin` searches for the array index corresponding to the smallest element (which is also the index of the element of x closest to y). We could write this code even more compactly as

```
z = x[argmin(abs(x - y))]
```

3.3.4 Array Slicing

We have emphasized that an array is laid out in memory in two pieces: a book-keeping block, and a sequence of storage blocks containing the elements of the array. Sometimes, it is possible for two arrays to share storage blocks. For instance, this happens when you perform array slicing:

```
>>> x = array([0.0, 1.0, 2.0, 3.0, 4.0, 5.0, 6.0])
>>> y = x[2:5]
>>> y
array([ 2.,  3.,  4.])
```

The statement $y = x[2:5]$ creates an array named y , containing a subset of the elements of x (i.e., the elements at indices 2, 3, and 4). However, Python does *not* accomplish this by copying the affected elements of x into a new array with new storage blocks. Instead, it creates a new book-keeping block for y , and points it towards the existing storage blocks of x :

Figure 3.3.2: Two arrays, x and y , sharing the same storage blocks.

Because the storage blocks are shared between two arrays, if we change an element in x , that effectively changes the contents of y as well:

```
>>> x[3] = 9.  
>>> y  
array([ 2.,  9.,  4.])
```

(The situation is similar if you specify a "step" during slicing, like `y = x[2:5:2]` . What happens in that case is that the data block keeps track of the step size, and Python can use this to figure out exactly which address to jump for accessing any given element.)

The neat thing about this method of sharing storage blocks is that slicing is an $O(1)$ operation, independent of the array size. Python does not need to do any copying on the stored elements; it merely needs to create a new book-keeping block. Therefore, slicing is a very "cheap" and efficient operation.

The downside is that it can lead to strange bugs. For example, this is a common mistake:

```
>>> x = y = linspace(0, 1, 100)
```

The above statement creates two arrays, `x` and `y` , pointing to the *same* storage blocks. This is almost definitely not what we intend! The correct way is to write two separate array initialization statements.

Whenever you intend to copy an array and change its contents freely without affecting the original array, you must remember to use the `copy` function:

```
>>> x = array([0.0, 1.0, 2.0, 3.0, 4.0, 5.0, 6.0])  
>>> y = copy(x[2:5])  
>>> x[3] = 9.  
>>> y  
array([ 2.,  3.,  4.])
```

In the above example, the statement `y = copy(x[2:5])` explicitly copies out the storage blocks of `x` . Therefore, when we change the contents of `x` , the contents of `y` are unaffected.

Do not call `copy` too liberally! It is an $O(N)$ operation, so unnecessary copying hurts performance. In particular, the basic arithmetic operations don't affect the contents of arrays, so it is always safe to write

```
>>> y = x + 4
```

rather than `y = copy(x) + 4` .

This page titled [3.3: Arrays](#) is shared under a [CC BY-SA 4.0](#) license and was authored, remixed, and/or curated by [Y. D. Chong](#) via [source content](#) that was edited to the style and standards of the LibreTexts platform.