9 Arrays

We salute the coders, designers, and programmers already hard at work at their desks, and we encourage every student who can't decide whether to take that computer science class to give it a try.
Michael Bloomberg, former mayor, New York City

A **list** is an abstract data type that describes a data structure that stores ordered values. Lists usually have an operation for creating a new, empty list, testing if it is empty, prepending an item, appending an item, and accessing an element at an index. You are already familiar with Python lists, one of many different implementations of the list abstract data type, which is a type of array. In this chapter, you will learn more about arrays.

An **array** is a data structure that stores elements with indexes in a contiguous block of memory. Arrays are often homogeneous and static. A **homogeneous data structure** can hold only a single data type, such as integers or strings. A static data structure is a data structure you cannot resize after you create it. When you create an array in a low-level programming language like C, you decide how many elements of a particular data type you want to store in it. Then, your computer assigns a block of memory for your array based on the number of elements and how much memory one element of that data type requires. This block consists of items stored one after the other in your computer's memory.

A Python list is a **heterogeneous variable-length array**. A **variable-length array** is one with a size that can change after you create it. A **heterogeneous array** is one that can hold different types of data rather than just a single type. Guido van Rossum wrote Python in C, but Python hides the complexities of array creation and manipulation. Instead, it presents you with a list data structure you can use without worrying about assigning its length ahead of time or specifying what data types it can hold.

Figure 9.1 shows an example of how a computer stores an array in its memory.

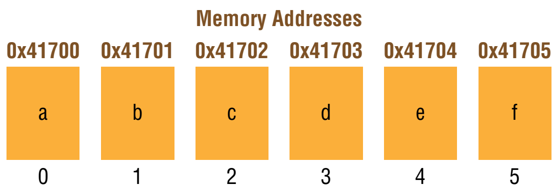

Figure 9.1: An example of data in an array

You can access each element in this array with a unique integer index. Usually, the first index in an array is index 0; however, different programming languages use different indexing schemes. Both Python and C use zero-based indexing, but some other languages, like MATLAB and Fortran, use one-based indexing (the first element is index 1). Some programming languages even allow you to use any integer value for the index of the first element. The memory location of the first element in an array is called its **base address**. When your computer needs to add a new item to an array, it calculates the location in memory it should put it in using this formula:

```
base_address + index * size_of_data_type
```

It takes the base address and adds it to the index multiplied by how much memory space the data type in the array takes up. Your computer also uses this formula when it needs to find an item in an array.

Arrays can be one-dimensional or multidimensional. In a **one-dimensional array**, you access each element in the array with an integer index:

```
array = [1, 2, 3]
print(array[0])

>> 1
```

In a **multidimensional array**, you access each individual element with two indexes (an integer index for each dimension):

```
multi_array = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
print(array[1][2])

>> 6
```

Regardless of whether an array is one-dimensional or multidimensional, your computer stores its elements in a contiguous block of memory and uses a mathematical formula that maps the index to a memory location to access them.

Array Performance

You can access and modify any single element in an array in constant time. For example, suppose you look up the data at index 3 in an array. In that case, your computer has to get information from only one memory location, even if there are a million elements in your array. Searching an unsorted array is O(*n*) because you need to check every item in the array. However, you can often sort arrays, like when you have a list of names or addresses, or a large set of numbers, in which case searching the array can be O(log *n*). Figure 9.2 shows the run times of arrays.

Data Structure	Time Complexity							
	Average				Worst			
	Access	Search	Insertion	Deletion	Access	Search	Insertion	Deletion
Array	O(1)	O(n)	O(n)	O(n)	O(1)	O(n)	O(n)	O(n)
Stack	O(n)	O(n)	O(1)	O(1)	O(n)	O(n)	O(1)	O(1)
Queue	O(n)	O(n)	O(1)	O(1)	O(n)	O(n)	O(1)	O(1)
Linked List	O(n)	O(n)	O(1)	O(1)	O(n)	O(n)	O(1)	O(1)
Hash Table	N/A	O(1)	O(1)	O(1)	N/A	O(n)	O(n)	O(n)
Binary Search Tree	O(log n)	O(log n)	O(log n)	O(log n)	O(n)	O(n)	O(n)	O(n)

Figure 9.2: Array operation run times

If arrays are so efficient, you might wonder why you shouldn't use them in every situation. While accessing and modifying individual elements in an array is very fast, modifying the shape of an array in any way (adding or deleting items) is O(*n*). Because your computer stores the elements in an array in a single continuous block of memory, inserting an element into an array means you have to shift all the elements coming after the added element, which is not efficient. For example, suppose you have an array that looks like Figure 9.3.

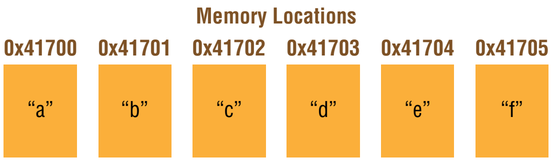

Figure 9.3: An array stored in a computer's memory

Look at what happens when you add *z* after *a* and *b* in Figure 9.4.

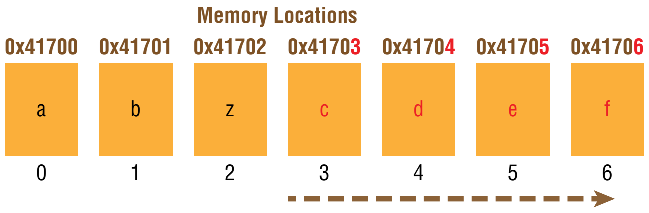

Figure 9.4: Adding data to an array often means changing many memory locations.

Making the third item in this list *z* requires your computer to shift four elements to different memory locations.

Having to shift elements is not a problem with small arrays, but if your program needs to add items to random locations in a large array (especially near the beginning), your computer may spend a lot of time copying memory around. This problem is worse for static arrays because your computer can't guarantee the memory addresses that come after the memory block it reserved for the array are free. That means when you add an element to an array in a language like C, you may need to reserve a new block of memory for the array, copy all the items from the old block to the new one, add the new element, and then free the old block. Python adds items to its lists more efficiently through a process called **overallocation**: reserving more memory for an array than it needs and keeping track of how many elements the array is storing and how much unused space it has.

As a programmer, you will frequently use arrays in your programs. You should consider using an array any time you need to store and access sequential data. For example, say you were programming a game like *Call of Duty* and you wanted to create a page that shows how the top 10 players rank. You could use an array to easily keep track of the top 10 players, their scores, and their order by storing the highest-ranking player at index 0 and the lowest ranking player at index 9. Arrays are one of the most important data structures for mathematical calculations. If you need to deal with large amounts of numerical data, you will get familiar with arrays. Computer scientists also use arrays to implement other data structures. For example, in later chapters you will learn how to implement stack and queue data structures using arrays.

Many programming languages' run-time systems use arrays to implement higher-level structures such as lists, which Python programmers use extensively. Arrays like the ones in Python's Numerical Python (NumPy) package are helpful for mathematical and scientific applications, financial applications, statistics, and so forth. NumPy arrays support mathematical operations such as matrix multiplication, which is used, for example, in graphics applications to scale, translate, and rotate graphical objects. In operating systems, arrays often handle any operation that manipulates a data sequence, such as memory management and buffering.

An array is not the best choice for large data sets you want to add data to often because adding items to an array is O(*n*). In this situation, linked lists, which you will learn about in the next chapter, are often a better choice. When you insert an item into an array, it changes the index of other elements, so if you need your data to keep the same index, a Python dictionary is probably a better choice.

Creating an Array

If you are programming in Python, you can use a list in most cases when you need an array. However, if you need the performance of a homogenous array, you can use Python's built-in `array` class. Here is how it works:

```
import array

arr = array.array('f', (1.0, 1.5, 2.0, 2.5))
print(arr[1])

>> 1
```

First, you import Python's built-in `array` module:

```
import array
```

Then, you pass `array.array` two parameters. The first parameter tells Python what data type you want your array to hold. In this case, `f` stands for float (a data type in Python for decimal numbers), but you can also create an array with Python's other data types. The second parameter is a Python list containing the data you want to put into your array.

```
arr = array.array('f', (1.0, 1.5, 2.0, 2.5))
```

Once you've created your array, you can use it like a Python list:

```
print(arr[1])
```

However, if you try to add a piece of data to your array other than the data type you initially passed in, you will get an error:

```
arr[1] = 'hello'

>> TypeError: "must be real number, not str"
```

Python's NumPy package also offers an array you can use to make calculations run nearly as fast as a lower-level programming language like C. You can learn more about creating arrays with NumPy by reading its documentation at `numpy.org`.

Moving Zeros

In a technical interview, you might have to locate all the zeros in a list and push them to the end, leaving the remaining elements in their original order. For example, suppose you have this list:

```
[8, 0, 3, 0, 12]
```

You would take it and return a new list with all the zeros at the end, like this:

```
[8, 3, 12, 0, 0]
```

Here is how you solve this problem in Python:

```
def move_zeros(a_list):
   zero_index = 0
   for index, n in enumerate(a_list):
       if n != 0:
           a_list[zero_index] = n
           if zero_index != index:
               a_list[index] = 0
           zero_index += 1
   return(a_list)

a_list = [8, 0, 3, 0, 12]
move_zeros(a_list)
print(a_list)
```

First, you create a variable called `zero_index` and set it to 0:

```
zero_index = 0
```

Then, you loop through every number in `a_list`, using the `enumerate` function to keep track of both the index and the current number in the list:

```
for index, n in enumerate(a_list):
```

Next comes this code, which executes only if `n` is not equal to 0:

```
if n != 0:
   a_list[zero_index] = n
   if zero_index != index:
       a_list[index] = 0
   zero_index += 1
```

When `n` is not equal to 0, you use the index stored in `zero_index` to replace whatever is at `zero_index` with `n`. Then, you check to see if `zero_index` and `index` are no longer the same number. If they are not the same number, it means there was a zero earlier in the list, so you replace whatever number is at the current index with 0 and increment `zero_index` by 1.

Let's take a look at why this works. Say this is your list, and your algorithm just hit the first zero, which means `index` is 1.

```
[8, 0, 3, 0, 12]
```

Because the number is a zero, this time around your loop, this code will not execute:

```
if n != 0:
    a_list[zero_index] = n
```

```
        if zero_index != index:
            a_list[index] = 0
        zero_index += 1
```

That means you did not increment `zero_index`. The next time around your loop, `index` is 2, `n` is 3, and your list is still the same.

```
[8, 0, 3, 0, 12]
```

Because this time `n` is not 0, this code executes:

```
if n != 0:
    a_list[zero_index] = n
    if zero_index != index:
        a_list[index] = 0
    zero_index += 1
```

When this part of your code runs:

```
a_list[zero_index] = n
```

it changes your list from this:

```
[8, 0, 3, 0, 12]
```

to this:

```
[8, 3, 3, 0, 12]
```

Then, this line of code:

```
if zero_index != index:
    a_list[index] = 0
```

changes your list from this:

```
[8, 3, 3, 0, 12]
```

to this:

```
[8, 3, 0, 0, 12]
```

Your code swapped the zero in the back of the list to the next nonzero number toward the front of the list.

Now, your algorithm hits a zero again, and the same thing happens.

```
[8, 3, 0, 0, 12]
```

Your variable `zero_index` and `index` are no longer the same and `zero_index` is 3, which is the index of the 0 farthest back in the list. The next time around, `n` is 12, so this code executes again:

```
if n != 0:
    a_list[zero_index] = n
    if zero_index != index:
        a_list[index] = 0
    zero_index += 1
```

This code:

```
a_list[zero_index] = n
```

changes your list from this:

```
[8, 3, 0, 0, 12]
```

to this:

```
[8, 3, 12, 0, 12]
```

This code:

```
if zero_index != index:
   a_list[index] = 0
```

changes it from this:

```
[8, 3, 12, 0, 12]
```

to this:

```
[8, 3, 12, 0, 0]
```

As you can see, the zeros are now at the end of the list, with the rest of the numbers preceding them in their original order.

Your algorithm contains one main loop that iterates through the elements in `a_list`, which means its time complexity is O(*n*).

Combining Two Lists

When preparing for a technical interview, you should be prepared to combine two lists, something you will have to do often in your day-to-day programming. For example, suppose you have a list of movies:

```
movie_list = [ "Interstellar", "Inception",
                "The Prestige", "Insomnia",
                "Batman Begins"
            ]
```

and a list of ratings:

```
ratings_list = [1, 10, 10, 8, 6]
```

You want to combine these two sets of data into a single list of tuples containing each movie title and its rating, like this:

```
[('Interstellar', 1),
('Inception', 10),
('The Prestige', 10),
('Insomnia', 8),
('Batman Begins', 6)]
```

You can use Python's built-in `zip` function to combine these lists:

```
print(list(zip(movie_list, ratings_list)))

>> [('Interstellar', 1), ('Inception', 10), ('The Prestige', 10), ('Insomnia', 8),
('Batman Begins', 6)]
```

The `zip` function takes one or more iterables and returns a `zip` object containing one piece of data from each iterable, which you then turn into a list. Your output is a list of tuples, with each list containing the name of a movie matched to its rating.

Finding the Duplicates in a List

Another common technical interview question is to check for duplicate items in a list, which you will also have to do often in real-world programming. One solution is to compare each item in your list to every other item. Unfortunately, comparing every item to every other item requires two nested loops and is O(*n***2). Python sets provide a better way to look for duplicates. A **set** is a data structure that cannot contain duplicate elements. If a set has a string such as `'Kanye West'`, it is impossible to add another instance of `'Kanye West'` to it.

Here is how to create a set and add data to it:

```
a_set = set()
a_set.add("Kanye West")
a_set.add("Kendall Jenner")
a_set.add("Justin Bieber")
print(a_set)

>> {'Kanye West', 'Kendall Jenner', 'Justin Bieber'}
```

Your code creates a set with three strings in it: "Kanye West", "Kendall Jenner", and "Justin Bieber". Now try to add a second instance of "Kanye West" to your set:

```
a_set = set()
a_set.add('Kanye West')
a_set.add('Kanye West')
a_set.add('Kendall Jenner')
a_set.add('Justin Bieber')
print(a_set)

>> {'Kanye West', 'Kendall Jenner', 'Justin Bieber'}
```

As you can see, your set still has only three items in it. Python did not add a second instance of 'Kanye West' because it is a duplicate.

Since sets cannot contain duplicates, you can add items from an iterable to a set one by one, and if its length does not change, you know the item you are trying to add is a duplicate.

Here is a function that uses sets to check for the duplicates in a list:

```
def return_dups(an_iterable):
   dups = []
   a_set = set()

   for item in an_iterable:
       l1 = len(a_set)
       a_set.add(item)
       l2 = len(a_set)
       if l1 == l2:
           dups.append(item)
   return dups

a_list = [
    "Susan Adams",
    "Kwame Goodall",
    "Jill Hampton",
    "Susan Adams"]

dups = return_dups(a_list)
print(dups)
```

The list you are evaluating contains four elements with one duplicate: "Susan Adams".

Your function return_dups accepts an iterable called an_iterable as a parameter:

```
def return_dups(an_iterable):
```

Inside your function, you create an empty list to hold the duplicates called dups:

```
dups = []
```

Then, you create an empty set called `a_set`:

```
a_set = set()
```

Next, you use a `for` loop to iterate through each item in `an_iterable`:

```
for item in an_iterable:
```

Next, you get the set's length, add an item from `an_iterable`, and check if the length changed:

```
l1 = len(a_set)
a_set.add(item)
l2 = len(a_set)
```

If your set's length did not change, the current item is a duplicate, so you append it to your `dups` list:

```
if l1 == l2:
    dups.append(item)
```

Here is your complete program:

```
def duplicates(an_iterable):
    dups = []
    a_set = set()
    for item in an_iterable:
        l1 = len(a_set)
        a_set.add(item)
        l2 = len(a_set)
        if l1 == l2:
            dups.append(item)
    return dups

a_list = [
    'Susan Adams',
    'Kwame Goodall',
    'Jill Hampton',
    'Susan Adams']

dups = duplicates(a_list)
print(dups)

>> ['Susan Adams']
```

When you run your function and pass in an iterable with duplicates, it outputs your `dups` list containing all the duplicates.

Finding the Intersection of Two Lists

Another common technical interview question involves writing a function to find the intersection of two lists, which is also helpful in your day-to-day programming. For example, say you have a list of winning lottery numbers in one list and another list that contains the most common winning lottery numbers of all time.

```
this_weeks_winners = [2, 43, 48, 62, 64, 28, 3]
most_common_winners = [1, 28, 42, 70, 2, 10, 62, 31, 4, 14]
```

Your goal is to find how many of the current winning numbers are in the winner's list.

One way to solve this problem is to use a list comprehension to create a third list and use a filter to check whether each value in `list1` is also in `list2`:

```
def return_inter(list1, list2):
    list3 = [v for v in list1 if v in list2]
    return list3

list1 = [2, 43, 48, 62, 64, 28, 3]
list2 = [1, 28, 42, 70, 2, 10, 62, 31, 4, 14]
print(return_inter(list1, list2))

>> [2, 62, 28]
```

As you can see, the numbers 2, 62, and 28 are in both lists.

This line of code uses a `for` loop to walk through `list1` and appends the item to your new list only if that value is also in `list2`.

```
list3 = [v for v in list1 if v in list2]
```

Python's `in` keyword searches an iterable for a value. Because you are dealing with unordered lists, Python performs a linear search when you use the keyword `in` inside your list comprehension. Since you are using the `in` keyword inside a loop (the first part of the list comprehension), your algorithm's time complexity is O(*n***2).

Another option is to use a set to solve this problem. In Python, sets have an `intersection` function that returns any elements that appear in two or more sets.

You can easily change lists into sets like this:

```
set1 = set(list1)
set2 = set(list2)
```

Once you've converted your lists to sets, you can apply the intersection function to find out where the two sets have duplicate items. Here is the syntax for calling the `intersection` function on two sets:

```
set1.intersection(set2)
```

Next, you convert the intersected set back into a list using the `list` function:

```
list(set1.intersection(set2))
```

Let's put it all together:

```
def return_inter(list1, list2):
    set1 = set(list1)
    set2 = set(list2)
    return list(set1.intersection(set2))

list1 = [2, 43, 48, 62, 64, 28, 3]
list2 = [1, 28, 42, 70, 2, 10, 62, 31, 4, 14]
new_list = return_inter(list1, list2)
print(new_list)

>> [2, 28, 62]
```

The first line defines a function called `return_inter` that accepts two lists as parameters:

```
def return_inter(list1, list2):
```

Next, you convert the lists into sets:

```
set1 = set(list1)
set2 = set(list2)
```

Then you call the `intersection` function and find the duplicates:

```
list(set1.intersection(set2))
```

Finally, you convert the set back into a list and return the result:

```
return list(set1.intersection(set2))
```

You are not limited to using the `intersection` function on only two sets. You can call it on as many sets as you like. This code finds the common elements of four sets:

```
(s1.intersection(s2, s3, s4))
```

Vocabulary

list: An abstract data type that describes a data structure that stores ordered values.

array: A data structure that stores elements with indexes in a contiguous block of memory.

homogeneous data structure: A data structure that can store elements of only a single data type, such as integer or float.

heterogeneous variable-length array: An array whose size can change after you create it that can also store multiple data types.

variable-length array: An array that's size can change after you create it.

heterogeneous array: An array that can hold different types of data.

base address: The memory location of the first element in an array.

one-dimensional array: An array where you access each element in the array by an integer index.

multidimensional array: An array where you access each element using an index tuple.

overallocation: Reserving more memory for a list than what it would strictly need and keeping track of how many elements the list is storing and how much unused space the list has.

set: A data structure that cannot contain duplicate elements.

Challenge

1. Given an array called `an_array` of non-negative integers, return an array consisting of all the even elements of `an_array`, followed by all the odd elements of `an_array`.