List is a versatile data structure and the principal data structure in Python. The data processed by a program is most commonly stored in a list, though there are situations where other data structures are preferable.

In this chapter, we take a look at the implementation and properties of the Python list. A particular focus is the time complexity of list operations: What operations are efficient and when you should use a list.

List in memory

The memory of a computer consists of a sequence of memory locations capable of storing data. Each memory location has an address that can be used for access. When a program is executed, the data it processes is stored in the memory.

Consider the following Python program as an example:

```
a = 7
b = -3
c = [1, 2, 3, 1, 2]
d = 99
```

Let us assume that the variables and the list defined in the program are stored in the memory starting from the address 100. The following is a simplified illustration of what the contents of the memory might look like in this situation:

100	101	102	103	104	105	106	107	108	109	110
7	-3	1	2	3	1	2	0	0	0	99
a	b	c								d

Here the contents of the variable `a` are stored in the memory location 100 and the contents of the variable `b` in the location 101. The memory locations 102–109 are reserved for the list `c`, but only the locations 102–106 are currently in use because the list has 5 elements. The contents of the variable `d` are in the location 110.

The elements of the list are stored in consecutive memory locations, which makes it easy to determine the location of a given list element. The memory address of an element is obtained by adding the element index to the address of the first element. For example in the above situation, the element `c[2]` is at the address 102 + 2 = 104.

As noted above, the list occupies more memory than is needed for its current elements. The reason for the extra memory is to prepare for possible addition of new elements to the list. Thus the list has two sizes: the number of elements (here 5) and the number of memory locations reserved for the list (here 8).

List operations

Python has several built-in operations for managing lists. We will next take a look at the efficiency of these operations from the time complexity perspective assuming that the list has n elements.

Knowing the time complexities of the list operations is important for algorithm design, because they determine which operations can be used as components of an efficient algorithm. Most list operations have one the following time complexities:

- $O(1)$: the operation is always efficient independent of the size of the list
- $O(n)$: the efficiency depends on the size of the list and the operation can be slow for large lists

Indexing

The elements of a list can be accessed and modified using the index operator `[]`.

```
numbers = [4, 3, 7, 3, 2]
print(numbers[2]) # 7
numbers[2] = 5
print(numbers[2]) # 5
```

The access or modification needs $O(1)$ time, because the elements are in consecutive memory locations and the address of an element can be computed quickly.

List size

The function `len` tells how many elements the list contains:

```
numbers = [4, 3, 7, 3, 2]
print(len(numbers)) # 5
```

The function needs $O(1)$ time, because the length is stored in memory with the list.

Searching

The operator `in` tells if a given element is on the list or not. The method `index` returns the index of the first occurrence of the element on the list. The method `count` counts the occurrences of the element on the list.

```
numbers = [4, 3, 7, 3, 2]

print(3 in numbers) # True
print(8 in numbers) # False

print(numbers.index(3)) # 1
print(numbers.count(3)) # 2
```

All of these operations need $O(n)$ time, because they have to scan through the list. For example, the method `count` operates essentially as the following function that uses a loop to go through the elements:

```
def count(items, target):
    result = 0
    for item in items:
        if item == target:
            result += 1
    return result
```

Notice that some of the operations can be fast in some situations. For example, if the element we are searching for is in the beginning of the list, the operator `in` is fast since it can stop immediately after finding the element. However, in the worst case the element is not on the list and the operation has to go through the whole list to verify this.

Adding an element

The method `append` adds an element to the end of the list, and the method `insert` adds an element to a given position on the list.

```
numbers = [1, 2, 3, 4]

numbers.append(5)
print(numbers) # [1, 2, 3, 4, 5]

numbers.insert(1, 6)
print(numbers) # [1, 6, 2, 3, 4, 5]
```

The time complexities of these operations are affected by the way the elements are stored in memory. In this case, the contents of the memory before additions look something like this:

100	101	102	103	104	105	106	107
1	2	3	4	0	0	0	0

The method `append` needs $O(1)$ time, because adding an element to the end of a list does not require changes to other memory locations. In the example, the element `5` is stored in the memory location 104:

100	101	102	103	104	105	106	107
1	2	3	4	5	0	0	0

If the memory area reserved for the list is already full and there is no room for the new element, more work is required. In such a case, a new bigger memory area is reserved for the list and all elements are moved to the new area, which needs $O(n)$ time. However, with a suitable implementation such a situation can be made so rare that the average time complexity of the `append` method is $O(1)$.

The method `insert` has time complexity $O(n)$ because, when an element is added to somewhere else than the end of the list, other elements need to be moved forward to make room for the new element. In our example, when the element `6` is inserted to the index `1`, the elements are relocated as follows:

100	101	102	103	104	105	106	107
1	6	2	3	4	5	0	0

Notice however, that the methdod `insert` is efficient when the element is inserted somewhere near the end of the list, because only a small number of elements needs to be relocated.

Removing an element

The method `pop` removes an element from a list. If the method is called without a parameter, it removes the last element. If a parameter is given, the operation removes the element at the specified index.

```
numbers = [1, 2, 3, 4, 5, 6]

numbers.pop()
print(numbers) # [1, 2, 3, 4, 5]

numbers.pop(1)
print(numbers) # [1, 3, 4, 5]
```

Let us again consider how removing an element affects the contents of the memory. Before the removes, the contents of the memory look something like this:

100	101	102	103	104	105	106	107
1	2	3	4	5	6	0	0

As with adding, removing an element at the end of a list takes $O(1)$ time, because the other elements are not affected. After removing the element `6', the memory might look like this:

100	101	102	103	104	105	106	107
1	2	3	4	5	0	0	0

Removing an element in the middle of a list needs $O(n)$ time, because now all the following elements have to be relocated in memory. Removing the element `2` has the following effect on the contents of the memory:

100	101	102	103	104	105	106	107
1	3	4	5	0	0	0	0

Python also has a list method `remove` that removes the first occurrence of a given element:

```
numbers = [1, 2, 3, 1, 2, 3]

numbers.remove(3)
print(numbers) # [1, 2, 1, 2, 3]
```

The time complexity of the method is always $O(n)$, because it first has to find the first occurrence (similarly to the method `index`), and then remove the element and relocate the following elements.

Summary

The following table summarizes the time complexities of the operations we have seen:

Operaatio	Aikavaativuus
Indexing (`[]`)	$O(1)$
Size (`len`)	$O(1)$
Is element on list? (`in`)	$O(n)$
Searching (`index`)	$O(n)$
Counting (`count`)	$O(n)$
Adding to end (`append`)	$O(1)$
Adding to middle (`insert`)	$O(n)$
Removing from end (`pop`)	$O(1)$
Removing from middle (`pop`)	$O(n)$
Searching and removing (`remove`)	$O(n)$

Thus the efficient operations are indexing, querying the size, and adding and removing elements at the end of the list. The list is a suitable data structure when most of the operations are among these, and the other more expensive operations are rare.

References and copying

In Python, lists and other data structures are accessed through references. Assigning a list to a variable only copies the reference, not the contents of the list:

```
a = [1, 2, 3, 4]
b = a
a.append(5)

print(a) # [1, 2, 3, 4, 5]
print(b) # [1, 2, 3, 4, 5]
```

Here executing the line `b = a` causes the variables `a` and `b` to refer to the same list in memory. When a new element is added to the list `a`, the same addition happens to the list `b`.

For copying the contents, we can use the method `copy`:

```
a = [1, 2, 3, 4]
b = a.copy()
a.append(5)

print(a) # [1, 2, 3, 4, 5]
print(b) # [1, 2, 3, 4]
```

Now the variables `a` and `b` refer to separate lists, and adding an element to the list `a` has no effect on the contents of the list `b`.

There is a big difference in the efficiency of the above operations: copying a reference takes $O(1)$ time, while copying the contents needs $O(n)$ time. Thus the line `b = a` takes $O(1)$ time, and the line `b = a.copy()` takes $O(n)$ time.

Side effects of functions

When a function is given a data structure as a parameter, only a reference is copied. Then the function can cause *side effects*, if it changes the contents of the data structure.

Consider the following function `double` that returns a list, where the value of each element has been doubled:

```
def double(numbers):
```

```
    result = numbers
    for i in range(len(result)):
        result[i] *= 2
    return result
```

Since only the reference is copied, the function changes the list it got as a parameter:

```
numbers = [1, 2, 3, 4]
print(double(numbers)) # [2, 4, 6, 8]
print(numbers) # [2, 4, 6, 8]
```

This is not good when the intention is to create a new list without changing the original list. The function can be corrected with the method `copy`:

```
def double(numbers):
    result = numbers.copy()
    for i in range(len(result)):
        result[i] *= 2
    return result
```

Now the contents of the original list are not changed:

```
numbers = [1, 2, 3, 4]
print(double(numbers)) # [2, 4, 6, 8]
print(numbers) # [1, 2, 3, 4]
```

Slicing and concatenation

The Python slice operator (`[:]`) creates a new list that contains a copy of a segment of the given list:

```
numbers = [1, 2, 3, 4, 5, 6, 7, 8]
print(numbers[2:6]) # [3, 4, 5, 6]
```

This operator needs $O(n)$ time because it copies the contents from the old list to the new list.

Since the slice operator copies the elements, it can be used for copying the contents of the whole list too. The following lines are equivalent:

```
result = numbers.copy()
```

```
result = numbers[:]
```

The operator `+` can be used for concatenating lists:

```
first = [1, 2, 3, 4]
second = [5, 6, 7, 8]
print(first + second) # [1, 2, 3, 4, 5, 6, 7, 8]
```

This takes $O(n)$ time, because the operator copies the elements from the original lists to the new list.

Lists in other languages

The list described in this chapter is more generally known as an *array list* or a *dynamic array*.

In low level languages (such as C++ and Java), the basic data structure is usually the *array*. Like a list, an array is a sequence of consecutive elements that can be accessed with indexing. However, an array is assigned a fixed memory area when it is created and its size cannot be changed later. When a variable size is required, these languages have other data structures.

In C++ the data structure `std::vector` implements a list:

```
std::vector<int> numbers;

numbers.push_back(1);
numbers.push_back(2);
numbers.push_back(3);
```

And in Java, the data structure `ArrayList` implements a list:

```
ArrayList<Integer> numbers = new ArrayList<>();
```

```
numbers.add(1);
numbers.add(2);
numbers.add(3);
```

In JavaScript, the basic data structure is called `Array`, but it is really a list since its size can be changed:

```
numbers = [];

numbers.push(1);
numbers.push(2);
numbers.push(3);
```