Python lists and arrays

We have used Python lists to store collections of objects. Lists are a quite powerful data structure: one can quite efficiently append elements at the end, and there are methods for inserting and removing elements at any position.

If you have programmed in Java, `C`, or `C++` before, you know that the only built-in method of storing collections of objects is as an array. If you need anything more sophisticated, you need to use a data structure from a library.

Python makes our life easier by providing Python lists as a basic data structure, but of course the Python interpreter has to implement this data structure. And, like in Java or `C`, it is implemented using an array. In this chapter we will study how this is done and how Python lists obtain their good performance.

Arrays

An array is just a block of memory, suitable to store references to a given, fixed number of objects. It is not possible to resize the memory block, so the array size remains fixed once the array has been created.

Since Python does not provide arrays, we will use a simple array simulation in the `cs206array` module. It provides the following:

- Create an array with n slots using `Array(n)`;
- Get the length of array `a` with `len(a)`;
- Access or modify the element at index i with `a[i]`;
- Loop over the elements of an array `a` with `for el in a`.

(The implementation in `cs206array.py` simply uses a list—but the point is that our array simulation will force us to work only with the operations available for arrays.)

Arrays that can grow

The main feature that distinguishes the Python list from an array is that it can grow and shrink when elements are added or removed. We will now try to implement such a data structure ourselves. We will call it `GrowArray` and we keep it simple: the only new operation we support is `append`, which adds an element at the end of the collection.

Consider example program `readwords2.py`. It creates an empty `GrowArray`, reads words from a large file, and appends each word to the `GrowArray`.

Here is our first attempt to define the `GrowArray`:

```
class GrowArray():
  def __init__(self):
    self._a = None

  def __len__(self):
    return len(self._a)

  def __getitem__(self, i):
    return self._a[i]

  def append(self, el):
    if self._a == None:
      self._a = Array(1)
      self._a[0] = el
    else:
      oldA = self._a
      n = len(oldA)
      self._a = Array(n + 1)
      for i in range(n):
```

```
      self._a[i] = oldA[i]
    self._a[n] = el
```

The `GrowArray` internally uses an array referenced from its field `_a`. (The underscore at the beginning of the field name indicates that clients of the `GrowArray` are not allowed to access this field directly. It is an implementation detail.)

The first time we append to an empty `GrowArray`, we create an array of size 1 and store the element there. When we append elements later, we first create a new array of the correct size (that is, the old size plus one), then we copy all elements from the old array to the new array, and finally we store the element `el` in the last slot of the new array. (Note that when `append` returns, there is no reference to the old array left, so it immediately becomes garbage.)

Let us analyze this technique. To keep the analysis simple, we will only count how often elements are copied from an old array to a new one. That is, we count how often the line `self._a[i] = oldA[i]` is executed. Clearly, when calling `a.append(el)`, this happends exactly `len(a)` times.

That means that when we store n words in a `GrowArray` using this method, the copy line is executed exactly

$$\sum_{k=0}^{n-1} k = \frac{n(n-1)}{2} \approx \frac{n^2}{2} \qquad \text{times.}$$

In other words, the total running time is quadratic in the number of words. We can indeed observe this behavior in our experiments: When we increase n by a factor 10, the running time increases roughly by a factor of 100.

How can we improve this? Clearly the problem is that creating a new array every time we append an element is very wasteful—we need to do this less often. The solution is to use an array that is *larger* than the current size of the `GrowArray`, so that we can add elements without having to recreate the array. Of course this means that we have to maintain the actual size of the `GrowArray` in a separate variable. Here is the new version (from example code `readwords3.py`):

```
class GrowArray():
  def __init__(self):
    self._a = Array(32)
    self._size = 0

  def __len__(self):
    return self._size

  def __getitem__(self, i):
    return self._a[i]

  def append(self, el):
    if self._size == len(self._a):
      # array is full, make a new one
      oldA = self._a
      n = len(oldA)
      self._a = Array(n + 32)
      for i in range(n):
        self._a[i] = oldA[i]
    self._a[self._size] = el
    self._size += 1
```

We store the actual size of the `GrowArray` in the field `_size`. Note that `len(a)` will return `a._zize`. In the beginning, the size is zero, even though the array `_a` can already hold 32 elements.

In the `append` method, we distinguish two cases: If `self._zize` is less than the length of the array `self._a`, then there is enough space left for the new element, so we can simply store it. We also need to increase `self._size` to reflect the new size of the `GrowArray`.

Otherwise, there is no space left to store the new element, so we need to create a new array. We make it large enough to store 32 extra elements, copy all elements from the old array to the new one, and finally store the new element and update the size.

Let's again count how often elements are copied from the old to the new array. This happens in `a.append(el)` exactly when `len(a)` is a multiple of 32, and then `len(a)` elements are copied. So, when appending n words in total, we execute this line

$$\sum_{k=1}^{\lfloor (n-1)/32 \rfloor} 32k \qquad \text{times.}$$

Let's set $m = \lfloor (n-1)/32 \rfloor$ (in Python syntax `m = (n-1) // 32`). Then the formula becomes

$$\sum_{k=1}^{m} 32k = 32 \sum_{k=1}^{m} k = 32 \frac{m(m+1)}{2} \approx 16m^2 \approx 16 \frac{n^2}{32^2} = \frac{n^2}{64}.$$

This is better than $n^2/2$, but only by a factor 32. The running time still grows quadratically in n, and again we can observe that in our experiments.

How can we do better than this? Changing the number 32 for a larger number only changes the constant factor, it doesn't change the quadratic behavior (do the calculation yourself!).

The solution is to make the increase larger as the size of the array gets larger. For instance, we can double the size of the array whenever it overflows, like this:

```
def append(self, el):
  if self._size == len(self._a):
    # array is full, make a new one
    oldA = self._a
    n = len(oldA)
    self._a = Array(2 * n)
    for i in range(n):
      self._a[i] = oldA[i]
  self._a[self._size] = el
  self._size += 1
```

(The only thing that has changed is the computation of the new array size.)

Again, we analyze the number of copy instructions. `a.append(el)` needs to enlarge the array every time that `len(a)` is 32, 64, 128, 256, etc., and then it makes `len(a)` copies. Let 2^m be the largest power of two that is less than n. Then the total number of copy instructions is

$$\sum_{k=5}^{m} 2^k = 2^{m+1} - 32 < 2n.$$

Our experiments show that indeed the running time increases as a linear function of n.

The disadvantage, of course, is that this method uses quite a bit of extra memory: If you need to store one million elements, you will actually create an array with two million slots. We can improve this by changing the factor two to something smaller, say 1.5 or 1.25.

Python lists

Python lists are implemented with exactly this method. You can use the function `slots` from the `cs206mem` module to determine the current capacity of a Python list. A little bit of experimentation (see example code `measure1.py`, `measure2.py`, and `measure3.py`) shows that when you append to a Python list that is completely filled, then a new array with $\lfloor 1.125(n+1) \rfloor + 6$ slots is created, where n is the previous array size.

Useful to know: when you create a list of a specific size, for instance using the expression `[ None ] * 1000` or `[0] * 1000`, the list will be created without any extra space, so the capacity will be equal to the size.