CS206A: Data Structures

2017-09-08

카이스트 전산학부 정지원(Otfried Cheong) 著
부경대학교 컴퓨터공학과 김태원 譯

2022년 6월 28일

파이썬 리스트와 배열

지금껏 파이썬 리스트로 객체 모음을 저장했다. 리스트는 꽤 강력한 자료 구조다. 리스트 끝에 요소(element)를 효율적으로 추가(append)할 수도 있다. 임의 위치 요소 삽입 및 제거 메소드도 존재한다.

Java, C, C++로 프로그래밍해 봤으면 알 테다. 객체 모음 저장용 내장형 메소드란 배열이 유일하다. 더 정교한 메소드가 필요하면 라이브러리에서 자료 구조를 사용해야 한다.

파이썬은 파이썬 리스트를 기본 자료 구조로 제공한다. 덕분에 우리 삶이 쉬워졌다. 다만 파이썬 리스트는 파이썬 인터프리터가 구현해야 한다. 물론이다. 그리고 Java나 C처럼 파이썬 리스트도 배열로 구현한다. 오늘은 배열을 통한 파이썬 리스트 구현 방식에 관해 학습한다. 또 파이썬 리스트가 좋은 성능을 취하는 방식에 관해 학습한다.

배열

배열이란 단순히 말해 메모리 블록(block, 덩어리)이다. 주어진 고정 개수 객체 레퍼런스를 저장하기 적합하다. 메모리 블록은 크기 재조정이 불가능하다. 그래서 배열은 생성 이후 고정 크기를 유지한다.

파이썬은 배열을 제공하지 않는다. 그러므로 `cs206array` 모듈에 있는 간단한 배열 시뮬레이션을 이용하겠다. `cs206array` 모듈에 있는 간단한 배열 시뮬레이션은 다음을 제공한다.

- `Array(n)`으로 n칸 배열 생성
- `len(a)`로 `a` 배열 길이 취득
- `a[i]`로 인덱스 i에 있는 요소 접근 혹은 조정
- `for el in a`로 `a` 배열 요소 루프

(`cs206array.py` 구현은 리스트를 사용할 따름이다. 하지만 요점은 따로 있다. 배열 시뮬레이션은 배열만 할 수 있는 연산을 강제한다.)

자랄 수 있는 배열

파이썬 리스트는 배열과 다르다. 원소를 추가하거나 제거할 때 파이썬 리스트는 자라고 쪼그라들 수 있다. 이제 자료 구조를 직접 구현해 보자. `GrowArray`라는 이름으로 단순하게 유지하겠다. 신규 지원 연산으로는 `append`가 유일하다. `append`는 모음 끝에 요소를 추가한다.

예제 프로그램 `readwords2.py`를 고찰하라. `readwords2.py`는 빈 `GrowArray`를 생성한다. 그런 다음 큰 파일에서 단어들을 읽는다. 그리하여 각 단어를 `GrowArray`에 추가한다.

아래는 `GrowArray`를 정의하려는 첫 시도다.

```
1  class GrowArray():
2      def __init__(self):
3          self._a = None
4
5      def __len__(self):
6          return len(self._a)
7
8      def __getitem__(self, i):
9          return self._a[i]
10
11     def append(self, el):
12         if self._a == None:
13             self._a = Array(1)
14             self._a[0] = el
15         else:
16             oldA = self._a
17             n = len(oldA)
18             self._a = Array(n + 1)
19             for i in range(n):
20                 self._a[i] = oldA[i]
21         self._a[n] = el
```

`GrowArray`는 `_a` 필드에서 참조한 배열을 내재적으로 이용한다. (필드명 앞쪽 `_`는 구현 디테일이다. `GrowArray` 클라이언트가 필드에 직접 접근할 수 없다는 암시다.)

빈 `GrowArray`에 처음 추가할 때는 크기 1 배열을 생성한다. 그런 다음 크기 1 배열에 요소를 저장한다. 나중에 요소를 추가할 때는 정확한 크기 (즉 원래 크기 더하기 1) 배열을 새롭게 생성한다. 그런 다음 원래 배열에서 새 배열로 모든 요소를 복사한다. 그리하여 새 배열 마지막 칸에 `el` 요소를 저장한다. (`append`가 반환할 때는 원래 배열에 대한 레퍼런스가 남지 않는다. 그래서 원래 배열은 즉각 가비지에 이른다. 유의하라.)

위 기법을 분석해 보자. 분석은 간단히 유지하겠다. 고로 원래 배열에서 새 배열로 요소를 복사한 빈도만 센다. 즉 `self._a[i] = oldA[i]` 라인 실행 빈도를 센다. `a.append(el)`을 호출할 때면 의심할 여지가 없다. 정확히 `len(a)`다.

`GrowArray`에 단어를 n개 저장하면 사본 라인이 정확히 아래만큼 실행된다는 말이다.

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

달리 말해 전체 실행 시간은 단어 개수에 대한 이차식이다. 실은 실험으로 관찰할 수 있는 행태다. n을 10배 증가하면 실행 시간은 대략 100배 증가한다.

어떻게 개선하겠는가? 분명 낭비가 문제다. 요소를 추가할 때마다 새 배열을 생성한다니. 낭비를 줄여야 한다. 해결책이 있다. GrowArray 현재 크기보다 큰 배열을 쓰겠다. 그리하여 배열을 재생성할 필요 없이 요소를 더할 수 있다. 물론 GrowArray 실제 크기를 변수에 따로 유지해야 한다는 뜻이다. 아래는 (예제 코드 readwords3.py에서 가져온) 새 판본이다.

```
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
```

_size 필드에 GrowArray 실제 크기를 저장한다. len(a)가 a._size를 반환한다. 유의하라. 예제 도입부에서 크기는 0이다. _a 배열이 이미 요소 32개를 취할 수 있는데도 말이다.

append 메소드는 두 가지 경우를 구별한다. 우선 self._size가 self._a 길이보다 작은 경우다. 그렇다면 새 요소를 위한 공간이 충분하다. 그래서 그냥 저장할 수 있다. 또 GrowArray 크기를 새롭게 반영하기 위해 self._size를 증가해야 한다.

두 번째는 self._size가 self._a 길이보다 크거나 같은 경우다. 그렇다면 새 요소를 위한 공간이 없다. 그래서 새 배열을 생성해야 한다. 추가 요소 32개를 저장하기 충분할 만큼 크게 만든다. 그런 다음 원래 배열에서 새 배열로 모든 요소를 복사한다. 그리하여 새로운 요소를 저장한다. 그리고 크기를 갱신한다.

원래 배열에서 새 배열로 요소를 몇 번 복사했는지 다시 센다. a.append(el)에서 len(a)가 32의 배수이고 len(a)개 요소를 복사할 때 빈도와 정확히 일치하는 횟수다. 그러므로 다해서 n개 단어를 추가할 때는 아래 라인을 실행한다.

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

$m = \lfloor (n-1)/31 \rfloor$이라고 (파이썬 구문으로는 m = (n-1) // 32) 두자. 식은 아래에 이른다.

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

$n^2/2$보다는 낫다. 하지만 32배 나을 뿐이다. 실행 시간은 여전히 n에 대한 이차식이다. 다시금 실험으로 관찰할 수 있는 행태다.

어떻게 개선할 수 있을까? 32를 더 큰 숫자로 바꿔봤자 상수 계수(係數, factor)만 바꿀 뿐이다. 이차식이라는 행태를 바꾸지는 않는다. (계산은 직접 하라!)

해결책이 있다. 배열 크기가 커질수록 증가율 또한 커지게 한다. 가령 오버플로가 일어날 때마다 배열 크기를 두 배로 늘릴 수 있다. 바로 이렇게.

```
    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
```

(신규 배열 크기 계산(computation)만 바꿨다.)

복사 명령 횟수를 다시 분석한다. `a.append(el)`은 매번 배열을 확대해야 한다. 그리하여 `len(a)`는 $32, 64, 128, 256, \ldots$에 이른다. 그리고 `a.append(el)`는 사본을 `len(a)`개 만든다. 2^m을 n보다 작은 수 가운데 가장 큰 2의 거듭제곱수로 둔다. 복사 명령 총 횟수는 아래와 같다.

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

실험은 실행 시간이 n에 대한 선형 함수(linear function)로 증가한다는 사실을 보인다.

물론 손실이 있다. 추가 메모리를 꽤 쓴다. 백만 요소를 저장해야 한다고 해 보자. 실제로는 이백만 칸 배열을 생성하기 마련이다. 개선할 수는 있다. 계수 2를 더 작은 수로 바꾼다. 예를 들어 1.5나 1.25.

파이썬 리스트

파이썬 리스트는 정확히 위 메소드(method, 방법)로 구현한다. 현재 파이썬 리스트 수용량(capacity)을 결정하고 싶을 수 있겠다. `cs206mem` 모듈에서 `slots` 함수를 사용하라. 약간의 실험 결과 하나. (예제 코드 `measure1.py, measure2.py, measure3.py`를 보라) 꽉 찬 파이썬 리스트에 추가(append)한다고 하자. 또 n을 원래 배열 크기라고 부르자. 그렇다면 $\lfloor 1.125(n+1) \rfloor + 6$ 칸 배열이 새롭게 생성된다.

특정 크기 리스트를 가령 `[ None ] * 1000` 혹은 `[0] * 1000`이라는 표현식으로 생성한다고 하자. 리스트는 아무 추가 공간 없이 생성되리다. 그래서 리스트 수용량은 리스트 크기와 서로 같으리다. 알아두면 좋다.