교수의 파이썬

04_4 append() 메소드와 + 연산자

창원대학교 정보통신공학과 교수 박동규

널널한 교수의 파이썬

04_4 append() 메소드와 + 연산자

창원대학교 정보통신공학과 교수 박동규

널널한 교수의 고급 파이썬

04_4 append() 메소드와 + 연산자

창원대학교 정보통신공학과 교수 박동규

널널한 교수의 고급 파이썬

04_4 append() 메소드와 + 연산자

창원대학교 정보통신공학과 교수 박동규

리스트 a에 새로운 원소 5를 추가하기...

```
a.append(5)     # 원소의 추가
```

리스트 a에 새로운 원소 5를 추가하기...

```
a.append(5)     # 원소의 추가
```

vs

리스트 a에 새로운 원소 5를 추가하기...

```
a.append(5)     # 원소의 추가
```

vs

```
a = a+[5]     # 원소의 추가
```

리스트 a에 새로운 원소 5를 추가하기...

```
a.append(5)     # 원소의 추가
```

vs

```
a = a+[5]     # 원소의 추가
```

어떤 차이가 있을까요?

리스트 a에 새로운 원소 5를 추가하기...

```
a.append(5)     # 원소의 추가
```

vs

```
a = a+[5]     # 원소의 추가
```

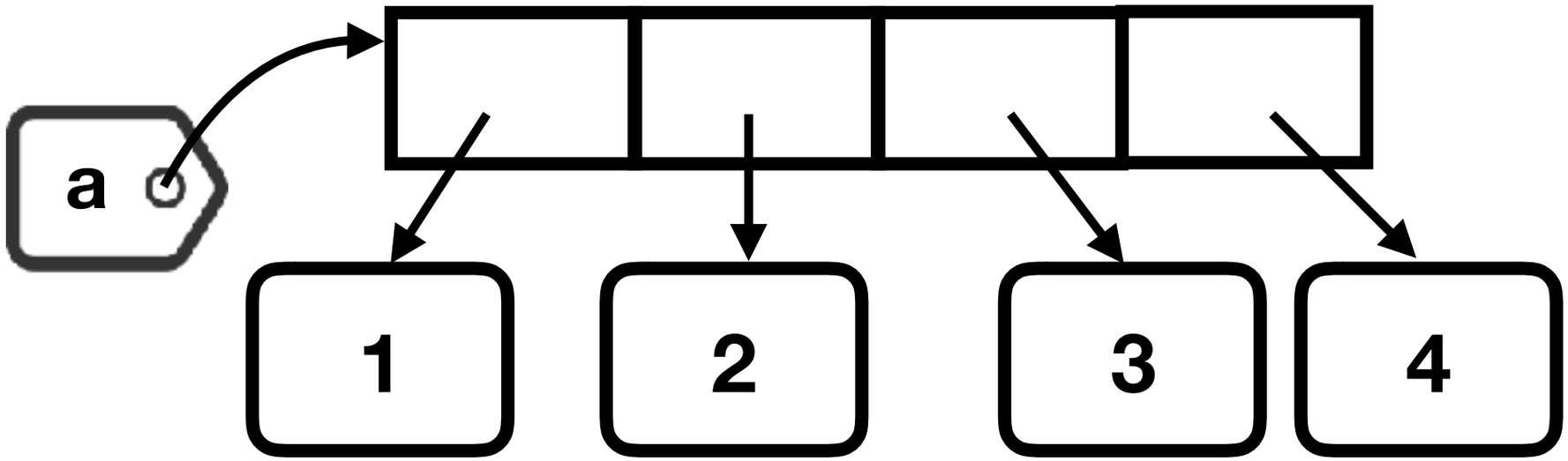
어떤 차이가 있을까요?

결과는 동일하지만 수행과정은 큰 차이가 있음

리스트 요소의 추가

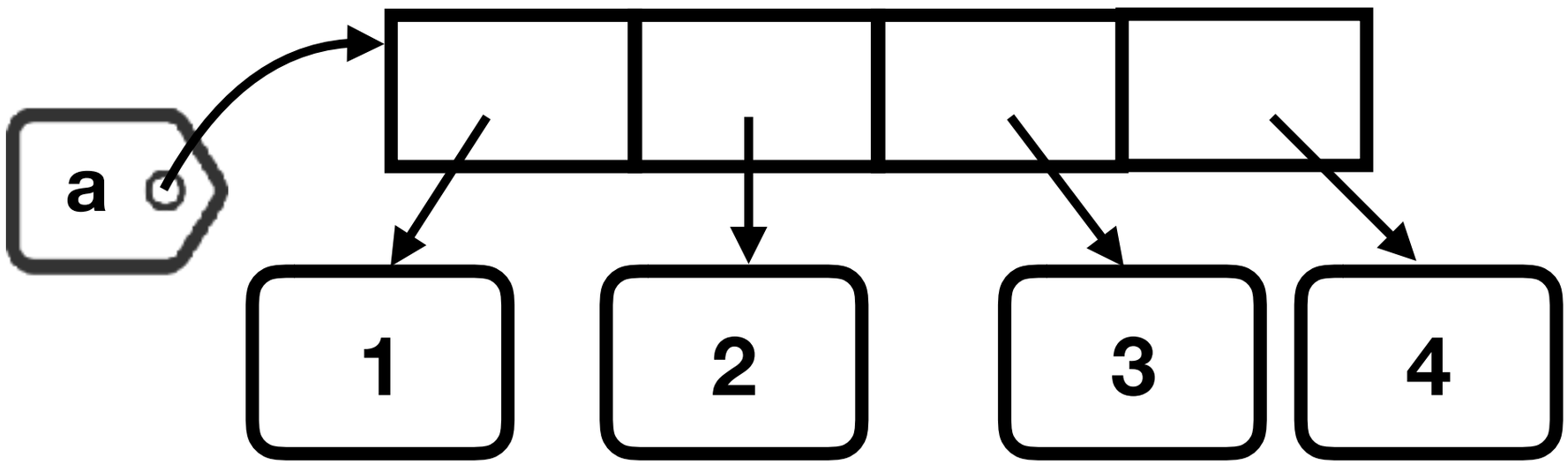
 # 리스트 객체 생성

a = [1, 2, 3, 4]



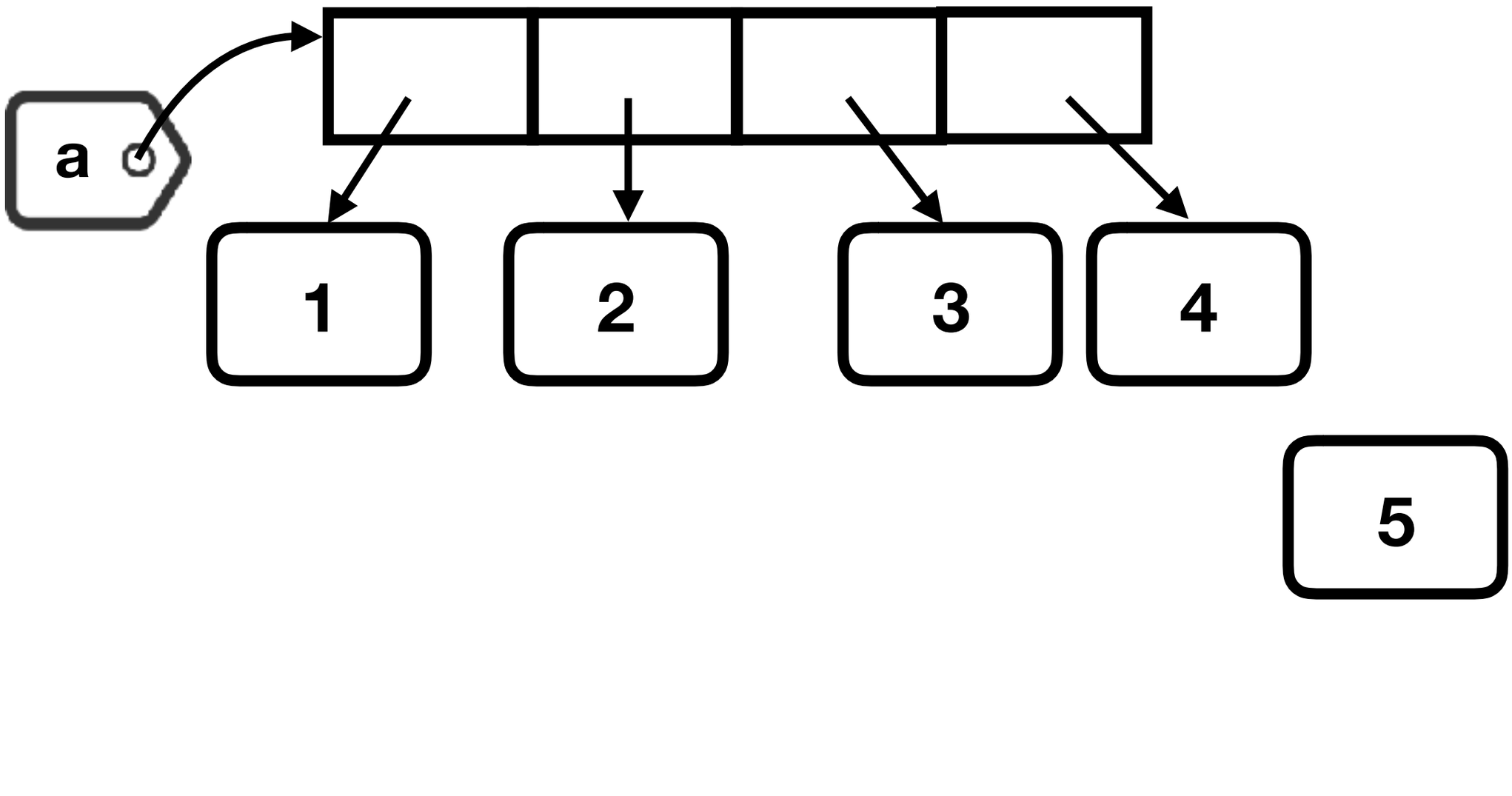
리스트 요소의 추가

```
# 리스트 객체 생성
a = [1, 2, 3, 4]
a.append(5)    # 리스트 객체의 변경(mutating)
```



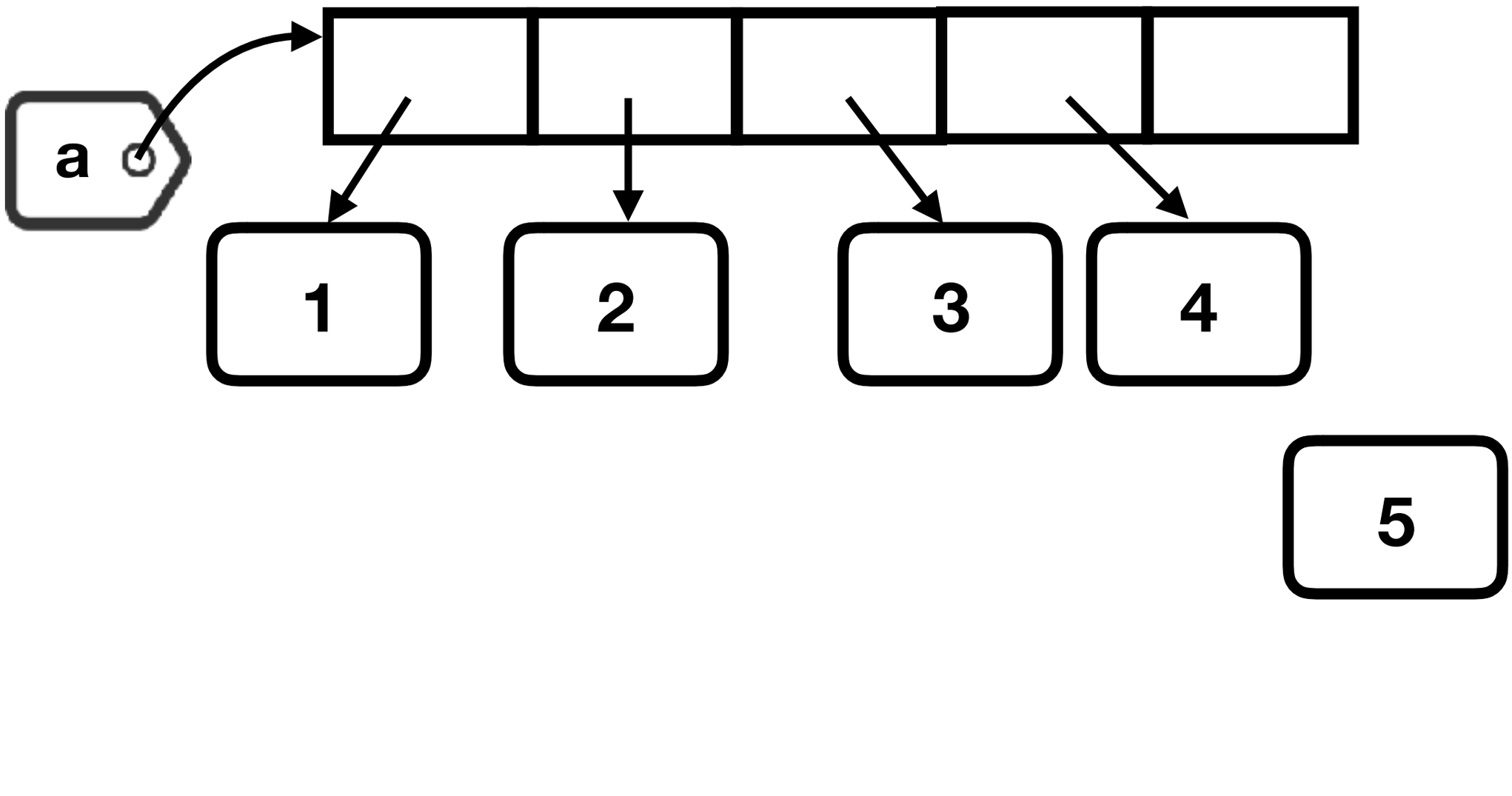
리스트 요소의 추가

```
# 리스트 객체 생성
a = [1, 2, 3, 4]
a.append(5)    # 리스트 객체의 변경(mutating)
```



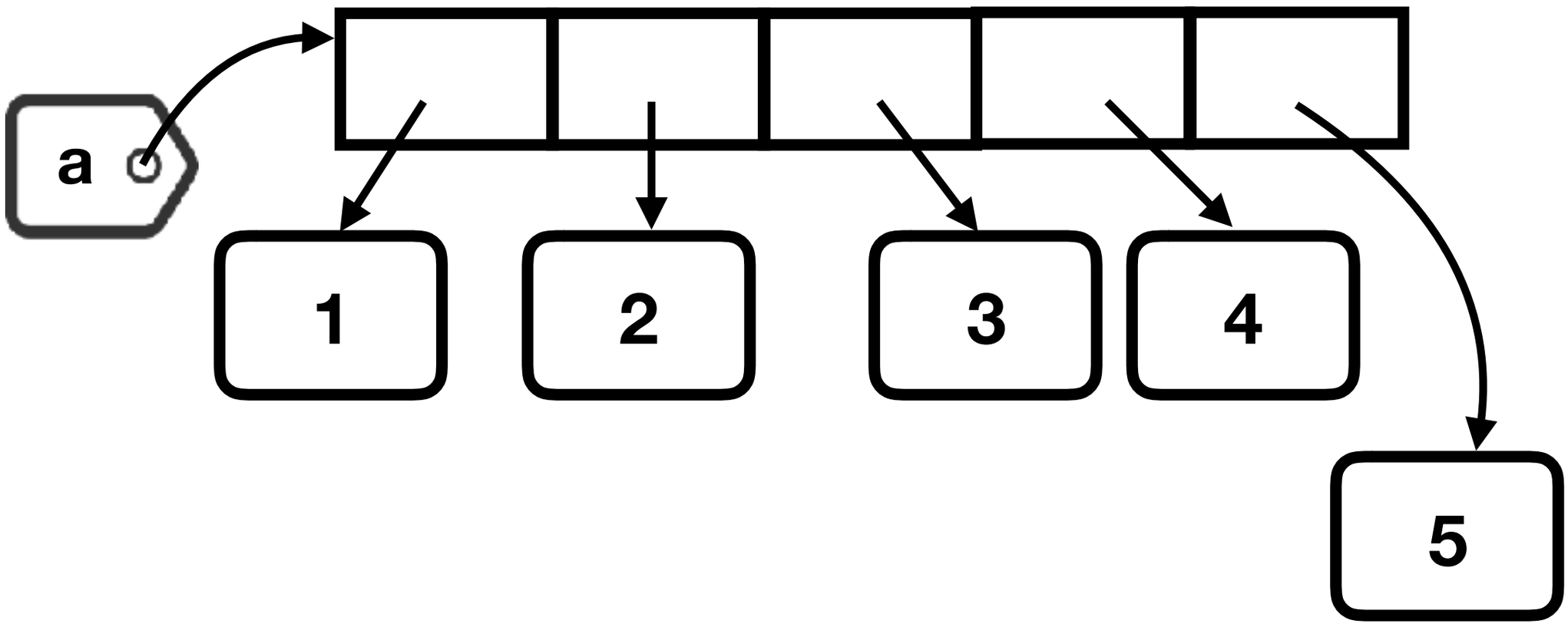
리스트 요소의 추가

```
# 리스트 객체 생성
a = [1, 2, 3, 4]
a.append(5)    # 리스트 객체의 변경(mutating)
```



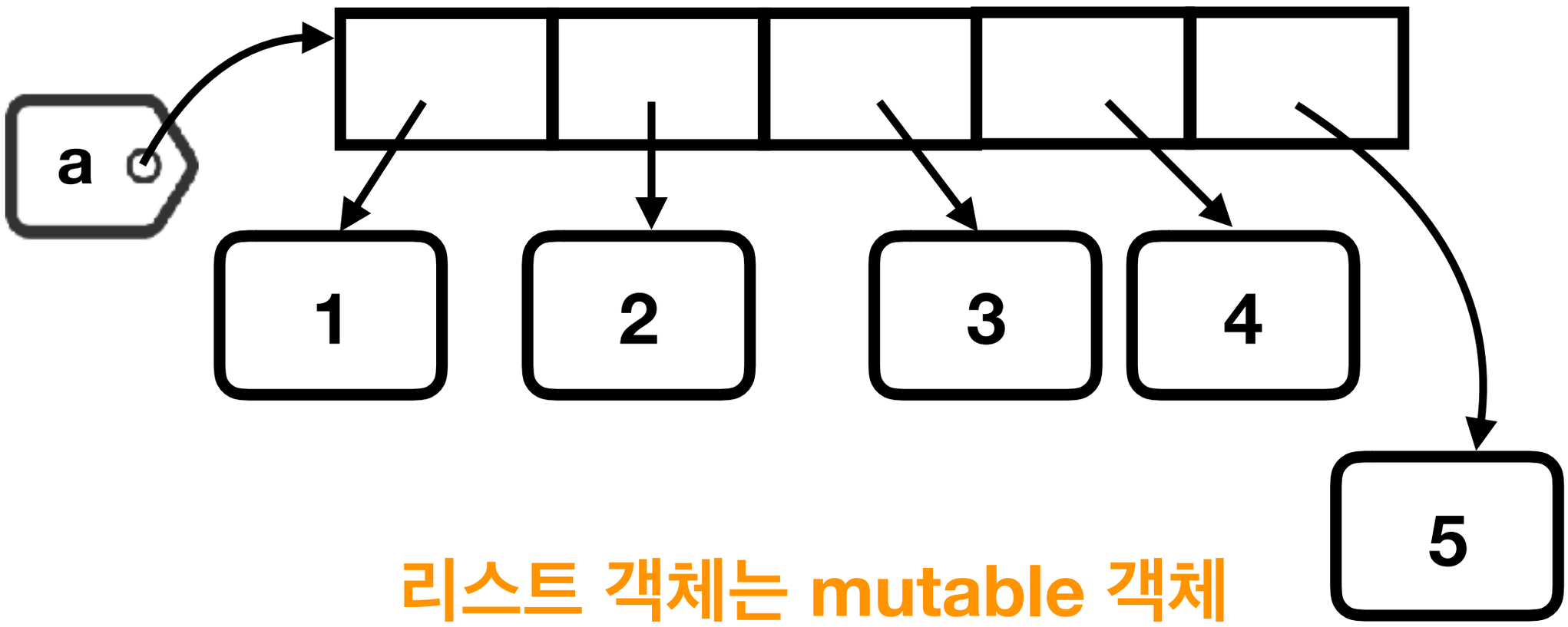
리스트 요소의 추가

```
# 리스트 객체 생성
a = [1, 2, 3, 4]
a.append(5)    # 리스트 객체의 변경(mutating)
```



리스트 요소의 추가

```
# 리스트 객체 생성
a = [1, 2, 3, 4]
a.append(5)    # 리스트 객체의 변경(mutating)
```

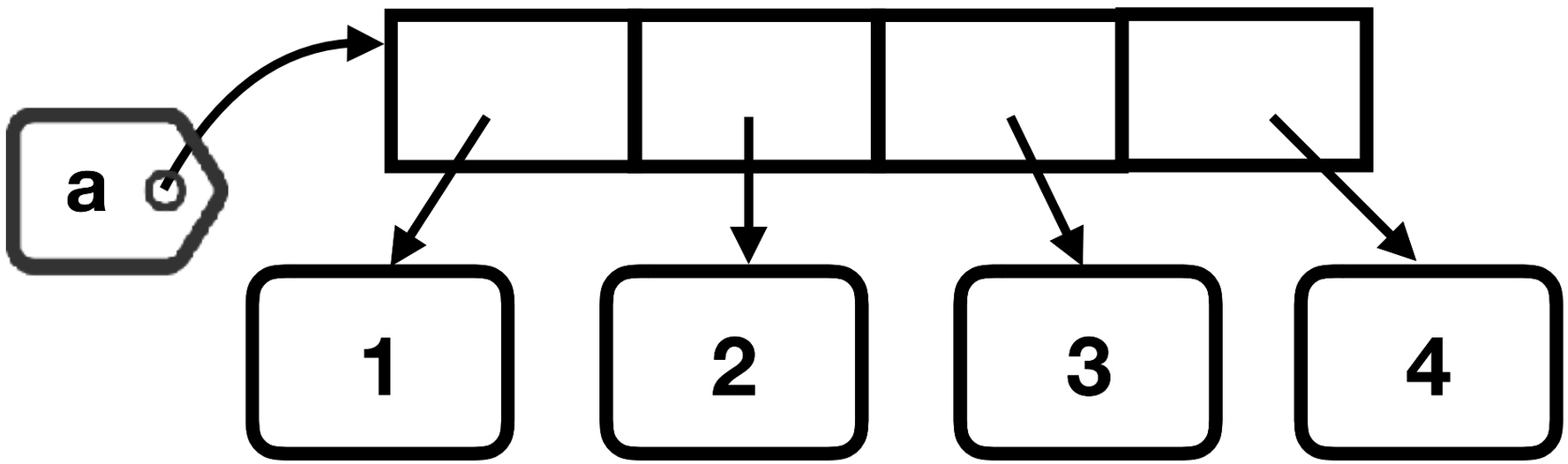


리스트 객체는 mutable 객체

리스트의 덧셈 연산과 재할당

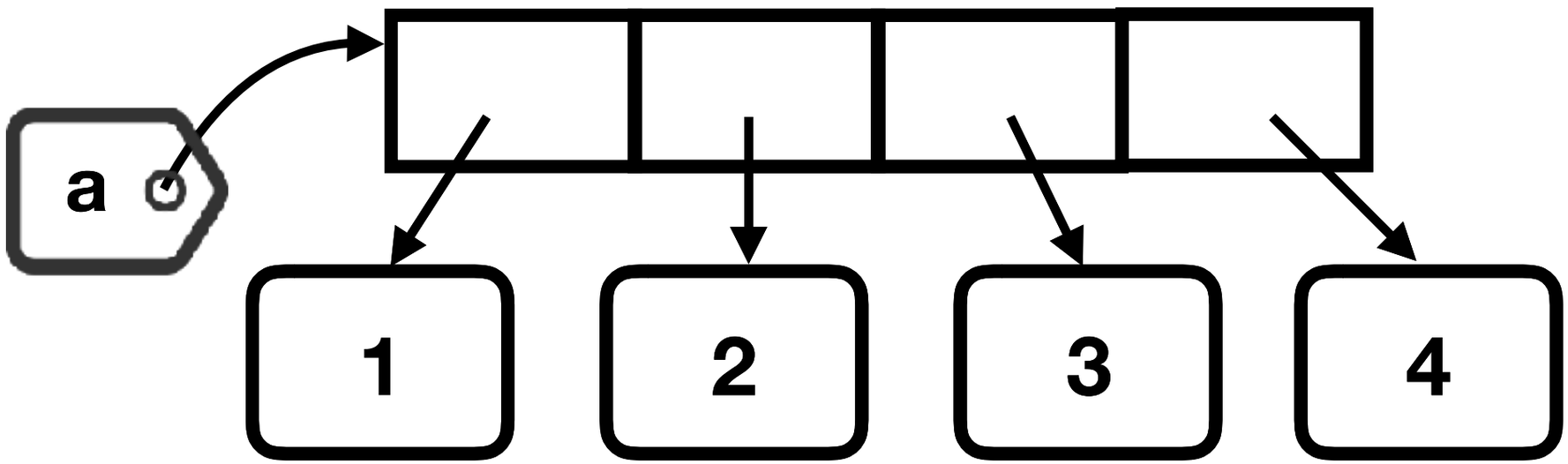
 # 리스트 객체 생성

a = [1, 2, 3, 4]



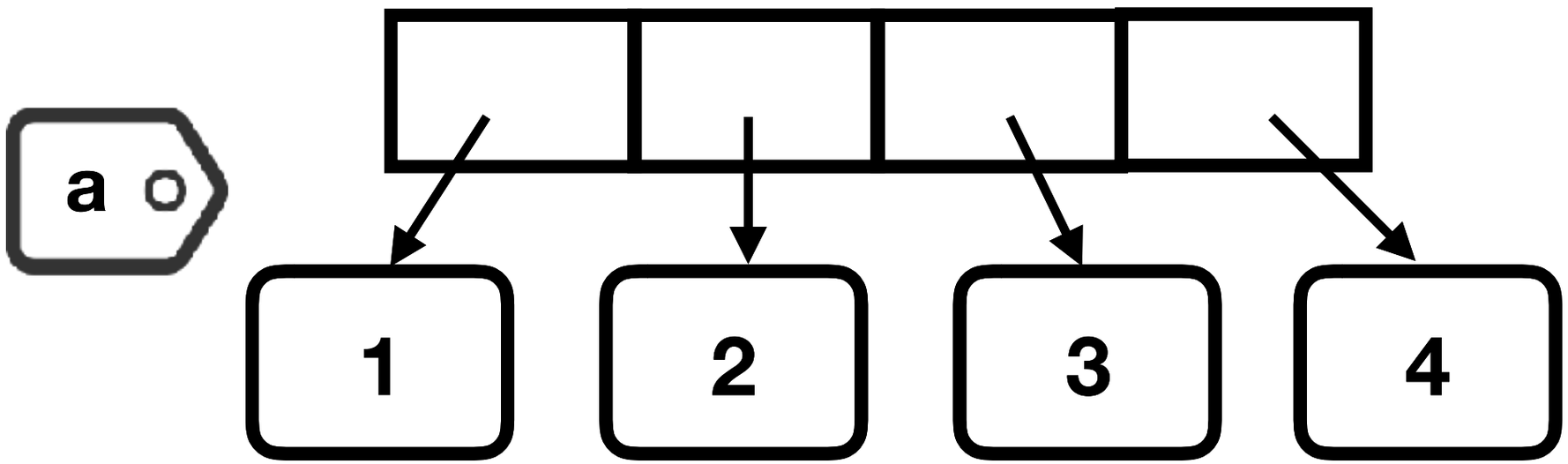
리스트의 덧셈 연산과 재할당

```
# 리스트 객체 생성
a = [1, 2, 3, 4]
a = a + [5]   # 리스트 객체의 재바인딩(rebinding)
```



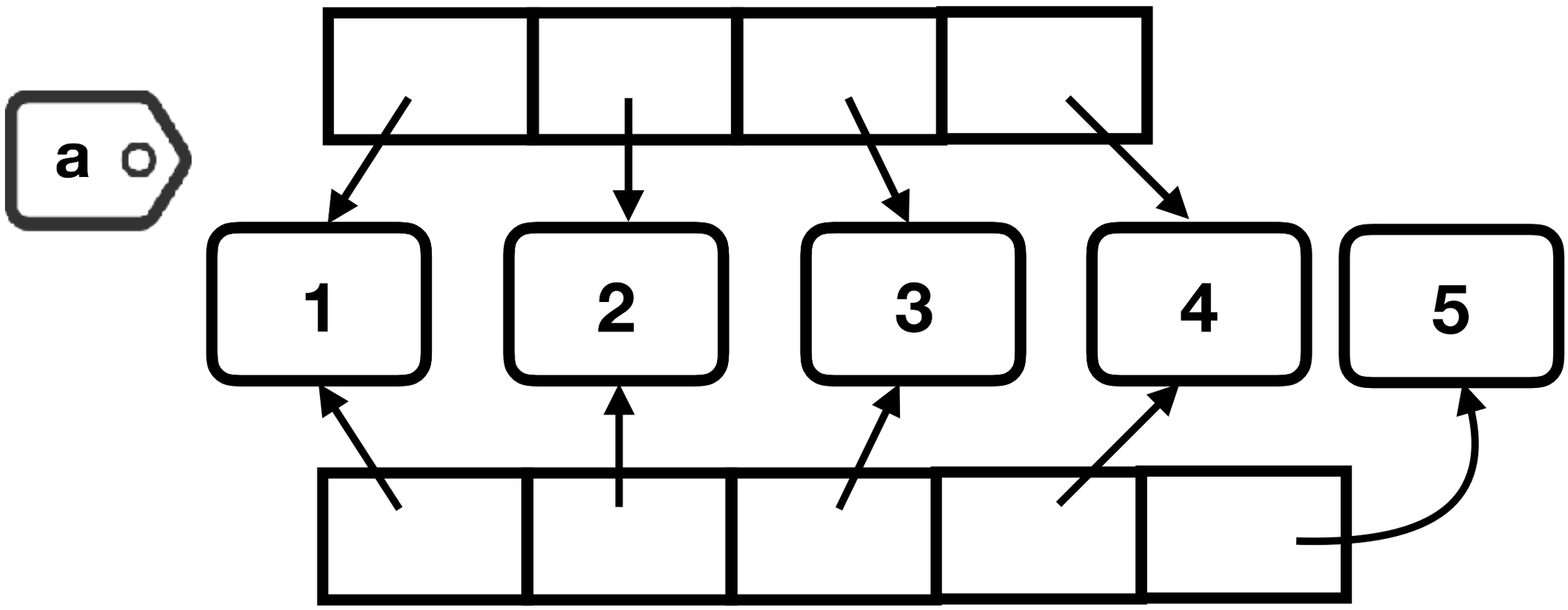
리스트의 덧셈 연산과 재할당

```
# 리스트 객체 생성
a = [1, 2, 3, 4]
a = a + [5]   # 리스트 객체의 재바인딩(rebinding)
```



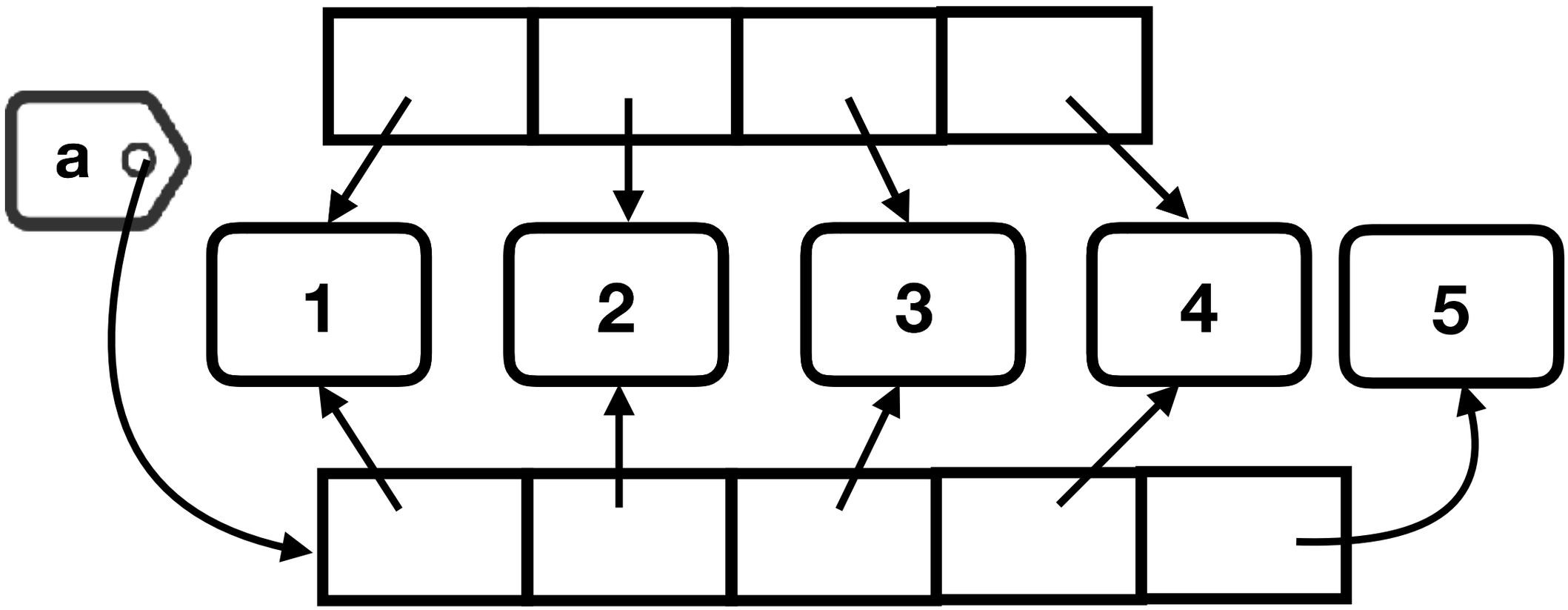
리스트의 덧셈 연산과 재할당

```
# 리스트 객체 생성
a = [1, 2, 3, 4]
a = a + [5]   # 리스트 객체의 재바인딩(rebinding)
```



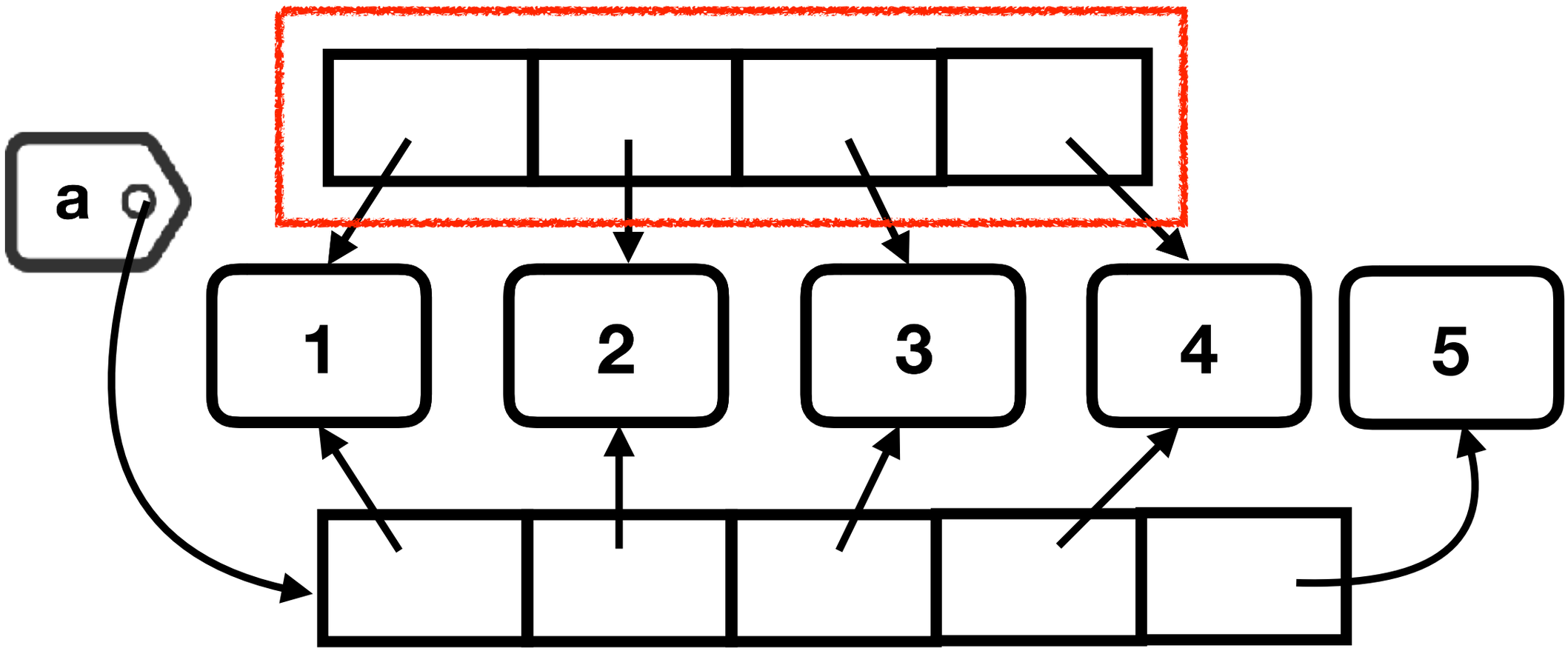
리스트의 덧셈 연산과 재할당

```
# 리스트 객체 생성
a = [1, 2, 3, 4]
a = a + [5]   # 리스트 객체의 재바인딩(rebinding)
```



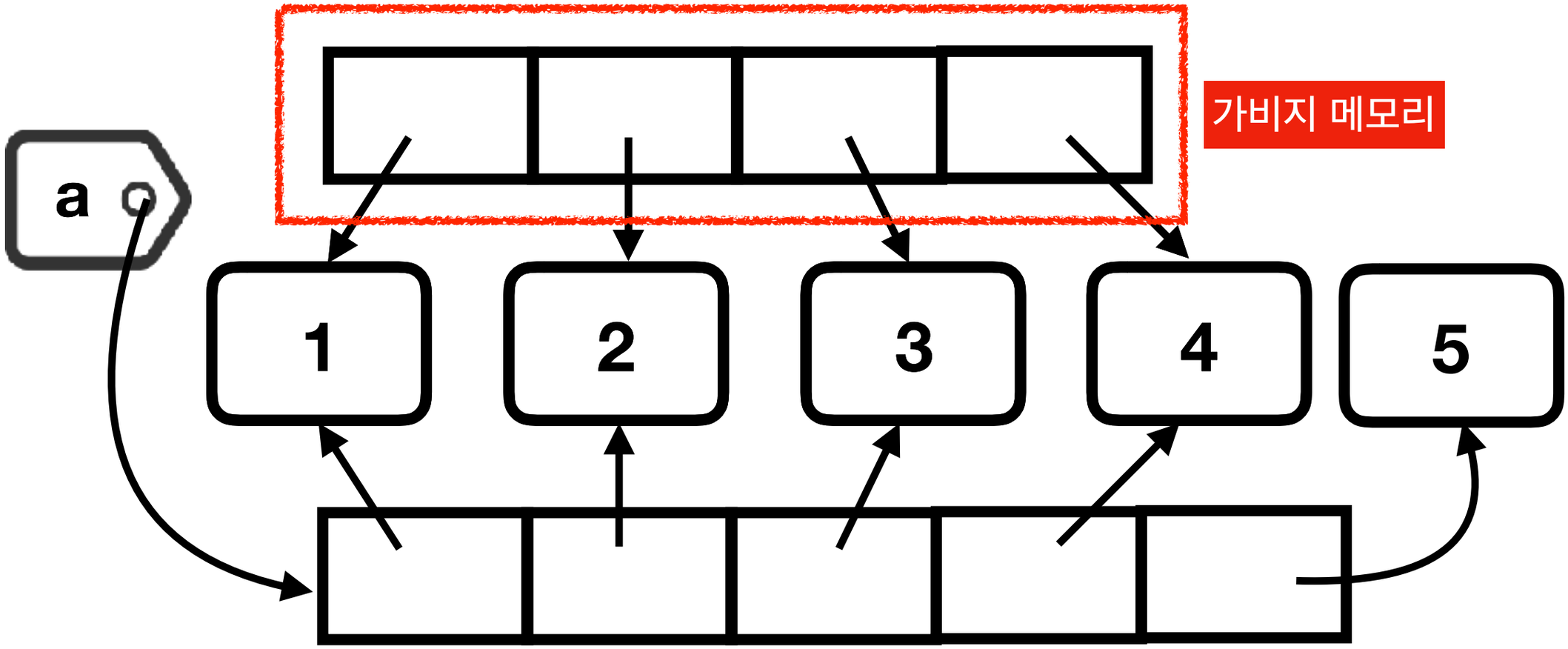
리스트의 덧셈 연산과 재할당

```
# 리스트 객체 생성
a = [1, 2, 3, 4]
a = a + [5]   # 리스트 객체의 재바인딩(rebinding)
```



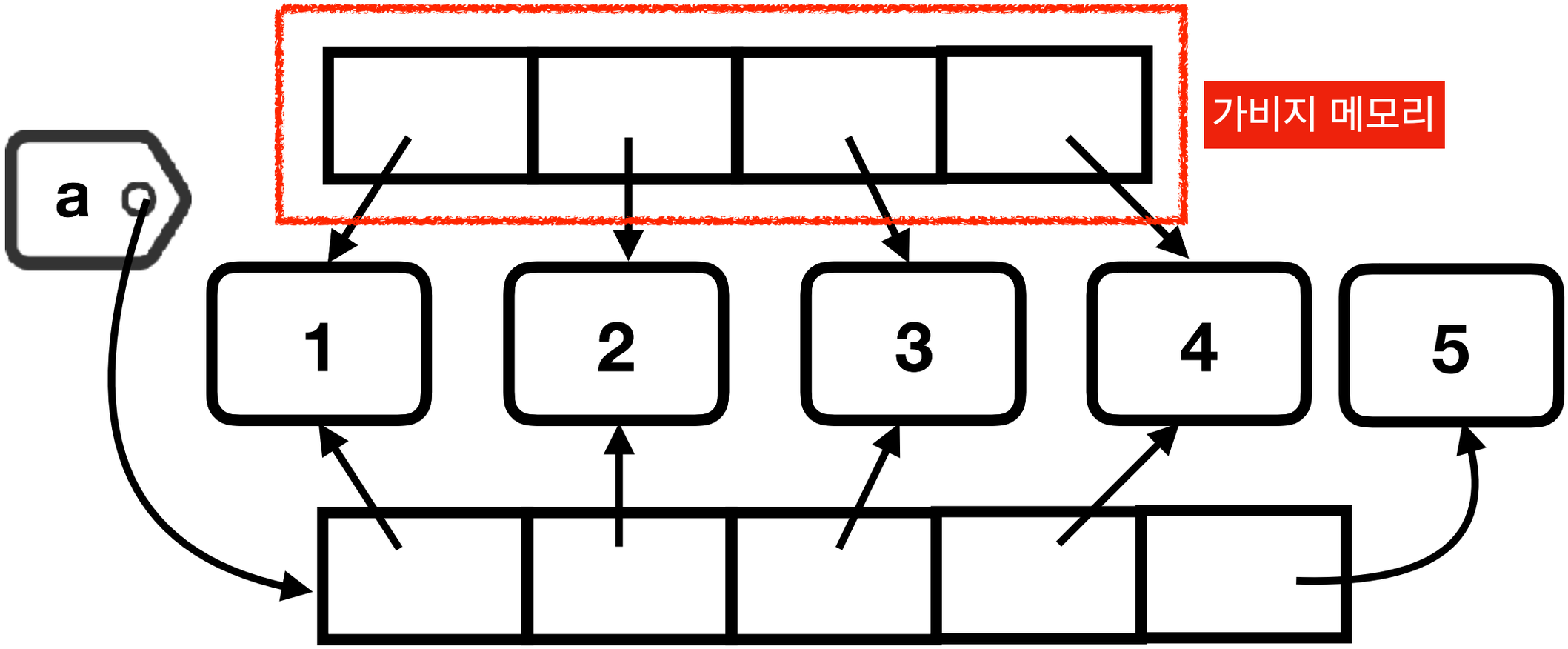
리스트의 덧셈 연산과 재할당

```
# 리스트 객체 생성
a = [1, 2, 3, 4]
a = a + [5]   # 리스트 객체의 재바인딩(rebinding)
```



리스트의 덧셈 연산과 재할당

```
# 리스트 객체 생성
a = [1, 2, 3, 4]
a = a + [5]   # 리스트 객체의 재바인딩(rebinding)
```



Lab

```
>>> a = [1, 2, 3, 4]
>>> id(a)
4512940232
>>> a.append(5)
>>> id(a)
4512940232
```

```
>>> a = [1, 2, 3, 4]
>>> id(a)
4512940552
>>> id(a[0])
4509013104
>>> a = a + [5]
>>> id(a)
4512940232
>>> id(a[0])
4509013104
```

Lab

```
>>> a = [1, 2, 3, 4]
>>> id(a)
4512940232
>>> a.append(5)
>>> id(a)
4512940232
```

```
>>> a = [1, 2, 3, 4]
>>> id(a)
4512940552
>>> id(a[0])
4509013104
>>> a = a + [5]
>>> id(a)
4512940232
>>> id(a[0])
4509013104
```

Lab

```
>>> a = [1, 2, 3, 4]
>>> id(a)
4512940232
>>> a.append(5)
>>> id(a)
4512940232
```

```
>>> a = [1, 2, 3, 4]
>>> id(a)
4512940552
>>> id(a[0])
4509013104
>>> a = a + [5]
>>> id(a)
4512940232
>>> id(a[0])
4509013104
```

Lab

```
>>> a = [1, 2, 3, 4]
>>> id(a)
4512940232
>>> a.append(5)
>>> id(a)
4512940232
```

append() 메소드의 수행 결과

```
>>> a = [1, 2, 3, 4]
>>> id(a)
4512940552
>>> id(a[0])
4509013104
>>> a = a + [5]
>>> id(a)
4512940232
>>> id(a[0])
4509013104
```

Lab

```
>>> a = [1, 2, 3, 4]
>>> id(a)
4512940232
>>> a.append(5)
>>> id(a)
4512940232
```

append() 메소드의 수행 결과

```
>>> a = [1, 2, 3, 4]
>>> id(a)
4512940552
>>> id(a[0])
 4509013104
>>> a = a + [5]
>>> id(a)
 4512940232
>>> id(a[0])
 4509013104
```

Lab

```
>>> a = [1, 2, 3, 4]
>>> id(a)
4512940232
>>> a.append(5)
>>> id(a)
4512940232
```

append() 메소드의 수행 결과

```
>>> a = [1, 2, 3, 4]
>>> id(a)
4512940552
>>> id(a[0])
4509013104
>>> a = a + [5]
>>> id(a)
4512940232
>>> id(a[0])
4509013104
```

Lab

```
>>> a = [1, 2, 3, 4]
>>> id(a)
4512940232
>>> a.append(5)
>>> id(a)
4512940232
```

append() 메소드의 수행 결과

```
>>> a = [1, 2, 3, 4]
>>> id(a)
4512940552
>>> id(a[0])
4509013104
>>> a = a + [5]
>>> id(a)
4512940232
>>> id(a[0])
4509013104
```

Lab

```
>>> a = [1, 2, 3, 4]
>>> id(a)
4512940232
>>> a.append(5)
>>> id(a)
4512940232
```

append() 메소드의
수행 결과

```
>>> a = [1, 2, 3, 4]
>>> id(a)
4512940552
>>> id(a[0])
 4509013104
>>> a = a + [5]
>>> id(a)
4512940232
>>> id(a[0])
4509013104
```

a = a + [5] 수행결과
a 객체는 다시 바인딩 된다

Lab

```
>>> a = [1, 2, 3, 4]
>>> id(a)
4512940232
>>> a.append(5)
>>> id(a)
4512940232
```

append() 메소드의
수행 결과

리스트 자료형은 가변(mutable)
= 객체의 내용이 바뀌어도 id는 안바뀜

```
>>> a = [1, 2, 3, 4]
>>> id(a)
4512940552
>>> id(a[0])
4509013104
>>> a = a + [5]
>>> id(a)
4512940232
>>> id(a[0])
4509013104
```

a = a + [5] 수행결과
a 객체는 다시 바인딩 된다

Lab

```
>>> a = [1, 2, 3, 4]
>>> id(a)
4512940232
>>> a.append(5)
>>> id(a)
4512940232
```

append() 메소드의
수행 결과

리스트 자료형은 가변(mutable)
= 객체의 내용이 바뀌어도 id는 안바뀜

```
>>> a = [1, 2, 3, 4]
>>> id(a)
4512940552
>>> id(a[0])
4509013104
>>> a = a + [5]
>>> id(a)
4512940232
>>> id(a[0])
4509013104
```

a = a + [5] 수행결과
a 객체는 다시 바인딩 된다

이것을 알아야 하는 이유

- 리스트 객체는 변경가능(mutable) 객체
  - int 형, tuple 형, str 형 객체는 변경불가능(immutable) 객체
- 리스트의 append() 메소드는 객체의 내용을 변경시킴
- 리스트의 + 연산은 객체로 복사해서 다시바인딩(리바인딩) 함
- 리바인딩은 시간이 많이 걸린다
- 리스트 객체의 append()는 상대적으로 빨리 수행된다

LAB

- 10만개의 데이터를 두 리스트에 삽입해 봅시다.
    - 한 번은 append() 메소드를 사용하고
    - 또 한 번은 b = b + [i] 와 같은 리바인딩을 사용해 봅시다
    - 마지막으로 list(range(100000)) 을 이용해 봅시다
- time 모듈의 time() 함수를 이용해서 시작 시간과 종료 시간을 기록한 다음 두 시간을 빼서 경과시간을 구합니다

[4]
```
import time

start_time = time.time()
a = []
for i in range(100000):
  a.append(i)

end_time = time.time()
t1 = end_time - start_time
print('append(i) 소요시간 = ', t1)
```

```
append(i) 소요시간 =  0.014089584350585938
```

[5]
```
start_time = time.time()
b = []
for i in range(100000):
  b = b + [i]

end_time = time.time()
t2 = end_time - start_time
print(' + [i]의 소요시간 = ', t2)
```

```
 + [i]의 소요시간 =  22.026045799255371
```

[6]
```
print('성능차이 {} 배'.format(t2/t1))
```

```
성능차이 1563.2857046162176 배
```

[4]
```
import time

start_time = time.time()
a = []
for i in range(100000):
  a.append(i)

end_time = time.time()
t1 = end_time - start_time
print('append(i) 소요시간 = ', t1)
```

```
append(i) 소요시간 =  0.014089584350585938
```

10만개의 데이터를 append() 메소드를 사용해서 리스트에 넣는데 0.014초가 소요됨

[5]
```
start_time = time.time()
b = []
for i in range(100000):
  b = b + [i]

end_time = time.time()
t2 = end_time - start_time
print(' + [i]의 소요시간 = ', t2)
```

```
 + [i]의 소요시간 =  22.026045799255371
```

[6]
```
print('성능차이 {} 배'.format(t2/t1))
```

```
성능차이 1563.2857046162176 배
```

```
[4]
```

```python
import time

start_time = time.time()
a = []
for i in range(100000):
  a.append(i)

end_time = time.time()
t1 = end_time - start_time
print('append(i) 소요시간 = ', t1)
```

```
append(i) 소요시간 =  0.014089584350585938
```

10만개의 데이터를 append() 메소드를 사용해서 리스트에 넣는데 0.014초가 소요됨

```
[5]
```

```python
start_time = time.time()
b = []
for i in range(100000):
  b = b + [i]

end_time = time.time()
t2 = end_time - start_time
print(' + [i]의 소요시간 = ', t2)
```

```
 + [i]의 소요시간 =  22.026045799255371
```

10만개의 데이터를 리스트에 넣는데 자그마치 22.026초가 소요됨
: b = b + [1]을 통해 매번 객체를 리바인딩 하기 때문에 속도가 느리며 가비지 메모리도 발생함

```
[6]
```

```python
print('성능차이 {} 배'.format(t2/t1))
```

```
성능차이 1563.2857046162176 배
```

```
[4]  1 import time
     2
     3 start_time = time.time()
     4 a = []
     5 for i in range(100000):
     6   a.append(i)
     7
     8 end_time = time.time()
     9 t1 = end_time - start_time
    10 print('append(i) 소요시간 = ', t1)
```

```
append(i) 소요시간 =  0.01408958435058593
```

10만개의 데이터를 append() 메소드를 사용해서 리스트에 넣는데 0.014초가 소요됨

```
[5]  1 start_time = time.time()
     2 b = []
     3 for i in range(100000):
     4   b = b + [i]
     5
     6 end_time = time.time()
     7 t2 = end_time - start_time
     8 print(' + [i]의 소요시간 = ', t2)
```

```
 + [i]의 소요시간 =  22.026045799255371
```

10만개의 데이터를 리스트에 넣는데 자그마치 22.026초가 소요됨
: b = b + [1]을 통해 매번 객체를 리바인딩 하기 때문에 속도가 느리며 가비지 메모리도 발생함

```
[6]  1 print('성능차이 {} 배'.format(t2/t1))
```

```
성능차이 1563.2857046162176 배
```

성능차이가 1563배

```
[4]  1 import time
     2
     3 start_time = time.time()
     4 a = []
     5 for i in range(100000):
     6   a.append(i)
     7
     8 end_time = time.time()
     9 t1 = end_time - start_time
    10 print('append(i) 소요시간 = ', t1)
```

```
append(i) 소요시간 =  0.01408958435058593
```

10만개의 데이터를 append() 메소드를 사용해서 리스트에 넣는데 0.014초가 소요됨

```
[5]  1 start_time = time.time()
     2 b = []
     3 for i in range(100000):
     4   b = b + [i]
     5
     6 end_time = time.time()
     7 t2 = end_time - start_time
     8 print(' + [i]의 소요시간 = ', t2)
```

```
 + [i]의 소요시간 =  22.026045799255371
```

10만개의 데이터를 리스트에 넣는데 자그마치 22.026초가 소요됨
: b = b + [1]을 통해 매번 객체를 리바인딩 하기 때문에 속도가 느리며 가비지 메모리도 발생함

```
[6]  1 print('성능차이 {} 배'.format(t2/t1))
```

```
성능차이 1563.2857046162176 배
```

성능차이가 1563배

```
[10]  1 start_time = time.time()
      2 c = list(range(100000))
      3 end_time = time.time()
      4 t3 = end_time - start_time
      5 print('list(range(100000)의 소요시간 = ', t3)
```

```
list(range(100000)의 소요시간 =  0.00445866584777832
```

```
[11]  1 print('성능차이 {} 배'.format(t2/t3))
```

```
성능차이 6675.450671087107 배
```

```
[10]  1 start_time = time.time()
      2 c = list(range(100000))
      3 end_time = time.time()
      4 t3 = end_time - start_time
      5 print('list(range(100000)의 소요시간 = ', t3)
```

list(range(100000)) 을 사용함

```
list(range(100000)의 소요시간 =  0.00445866584777832
```

```
[11]  1 print('성능차이 {} 배'.format(t2/t3))
```

```
성능차이 6675.450671087107 배
```

```
[10]  1 start_time = time.time()
      2 c = list(range(100000))
      3 end_time = time.time()
      4 t3 = end_time - start_time
      5 print('list(range(100000)의 소요시간 = ', t3)
```

list(range(100000)) 을 사용함

```
list(range(100000)의 소요시간 =  0.00445866584777832
```

```
[11]  1 print('성능차이 {} 배'.format(t2/t3))
```

성능차이가 자그마치 6675배

```
성능차이 6675.450671087107 배
```

```
[10]  1 start_time = time.time()
      2 c = list(range(100000))
      3 end_time = time.time()
      4 t3 = end_time - start_time
      5 print('list(range(100000)의 소요시간 = ', t3)
```

list(range(100000)) 을 사용함

```
list(range(100000)의 소요시간 =  0.00445866584777832
```

```
[11]  1 print('성능차이 {} 배'.format(t2/t3))
```

성능차이가 자그마치 6675배

```
성능차이 6675.450671087107 배
```

```
[10]  1 start_time = time.time()
      2 c = list(range(100000))
      3 end_time = time.time()
      4 t3 = end_time - start_time
      5 print('list(range(100000)의 소요시간 = ', t3)
```

list(range(100000)) 을 사용함

```
list(range(100000)의 소요시간 =  0.00445866584777832
```

```
[11]  1 print('성능차이 {} 배'.format(t2/t3))
```

성능차이가 자그마치 6675배

```
성능차이 6675.450671087107 배
```

```
[14]  1 import numpy as np
      2
      3 start_time = time.time()
      4 d = np.arange(100000)
      5 end_time = time.time()
      6 t4 = end_time - start_time
      7 print('list(range(100000)의 소요시간 = ', t4)
```

```
list(range(100000)의 소요시간 =  0.0008542537689208984
```

```
[15]  1 print('성능차이 {} 배'.format(t2/t4))
```

```
성능차이 34841.64192017862 배
```

```
[14]  1 import numpy as np
      2
      3 start_time = time.time()
      4 d = np.arange(100000)
      5 end_time = time.time()
      6 t4 = end_time - start_time
      7 print('list(range(100000)의 소요시간 = ', t4)
```

numpy를 사용하세요

```
list(range(100000)의 소요시간 =  0.0008542537689208984
```

```
[15]  1 print('성능차이 {} 배'.format(t2/t4))
```

```
성능차이 34841.64192017862 배
```

```
[14]  1 import numpy as np
      2
      3 start_time = time.time()
      4 d = np.arange(100000)
      5 end_time = time.time()
      6 t4 = end_time - start_time
      7 print('list(range(100000)의 소요시간 = ', t4)
```

numpy를 사용하세요

```
list(range(100000)의 소요시간 =  0.0008542537689208984
```

```
[15]  1 print('성능차이 {} 배'.format(t2/t4))
```

성능차이가 자그마치 34,841배

```
성능차이 34841.64192017862 배
```

```
[14]  1 import numpy as np
      2
      3 start_time = time.time()
      4 d = np.arange(100000)
      5 end_time = time.time()
      6 t4 = end_time - start_time
      7 print('list(range(100000)의 소요시간 = ', t4)
```

numpy를 사용하세요

```
list(range(100000)의 소요시간 =  0.0008542537689208984
```

```
[15]  1 print('성능차이 {} 배'.format(t2/t4))
```

성능차이가 자그마치 34,841배

```
성능차이 34841.64192017862 배
```

```
[14]  1 import numpy as np
      2
      3 start_time = time.time()
      4 d = np.arange(100000)
      5 end_time = time.time()
      6 t4 = end_time - start_time
      7 print('list(range(100000)의 소요시간 = ', t4)
```

numpy를 사용하세요

```
list(range(100000)의 소요시간 =  0.0008542537689208984
```

```
[15]  1 print('성능차이 {} 배'.format(t2/t4))
```

성능차이가 자그마치 34,841배

```
성능차이 34841.64192017862 배
```

```
[14]  1 import numpy as np
      2
      3 start_time = time.time()
      4 d = np.arange(100000)
      5 end_time = time.time()
      6 t4 = end_time - start_time
      7 print('list(range(100000)의 소요시간 = ', t4)
```

numpy를 사용하세요

```
list(range(100000)의 소요시간 =  0.0008542537689208984
```

```
[15]  1 print('성능차이 {} 배'.format(t2/t4))
```

성능차이가 자그마치
34,841배

```
성능차이 34841.64192017862 배
```

numpy의 ndarray 객체
- 동일 자료형의 객체를
- 이웃한 메모리 위치에 삽입
- C언어의 배열과 유사하다

```
[14]  1 import numpy as np
      2
      3 start_time = time.time()
      4 d = np.arange(100000)
      5 end_time = time.time()
      6 t4 = end_time - start_time
      7 print('list(range(100000)의 소요시간 = ', t4)
```

numpy를 사용하세요

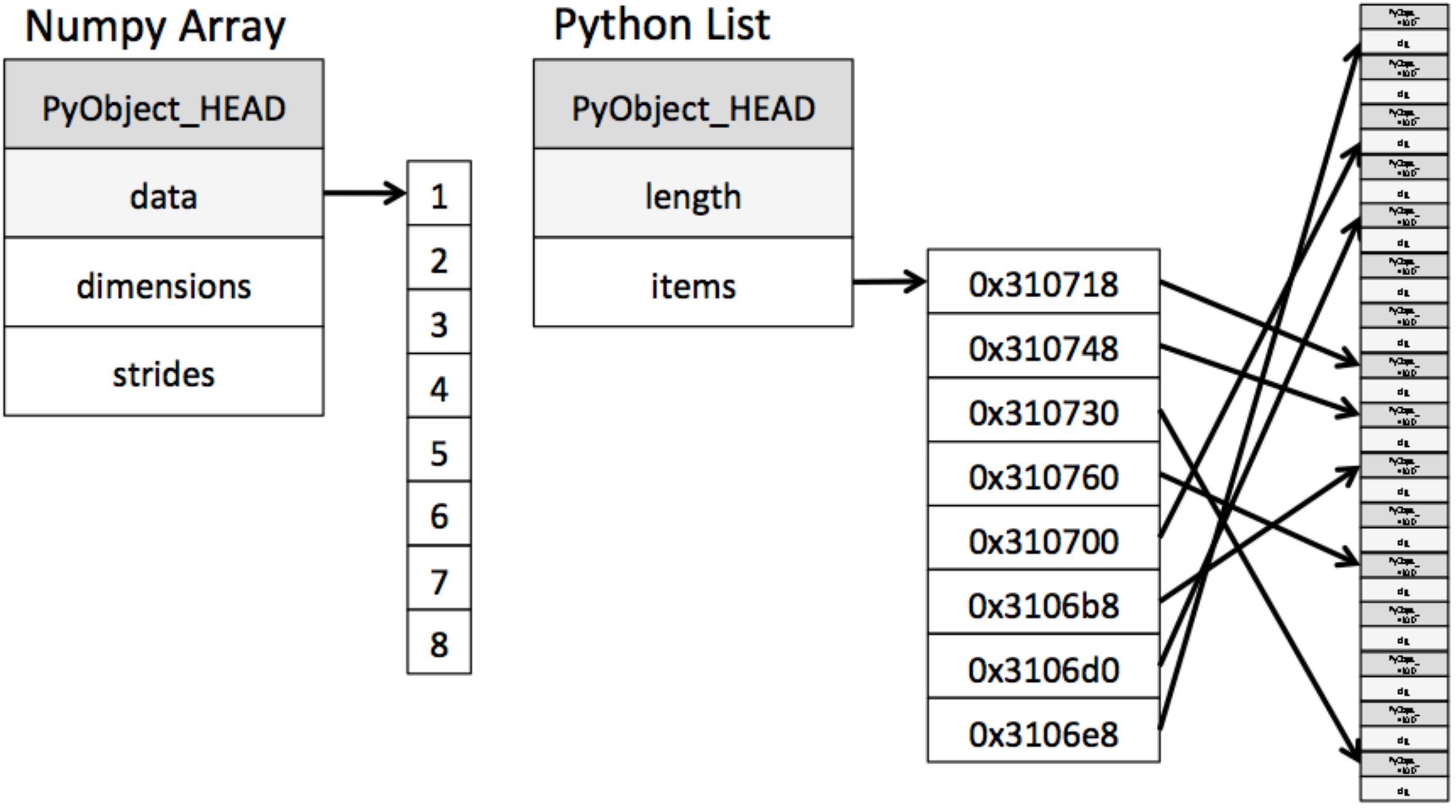
```
list(range(100000)의 소요시간 =  0.0008542537689208984
```

```
[15]  1 print('성능차이 {} 배'.format(t2/t4))
```

성능차이가 자그마치
34,841배

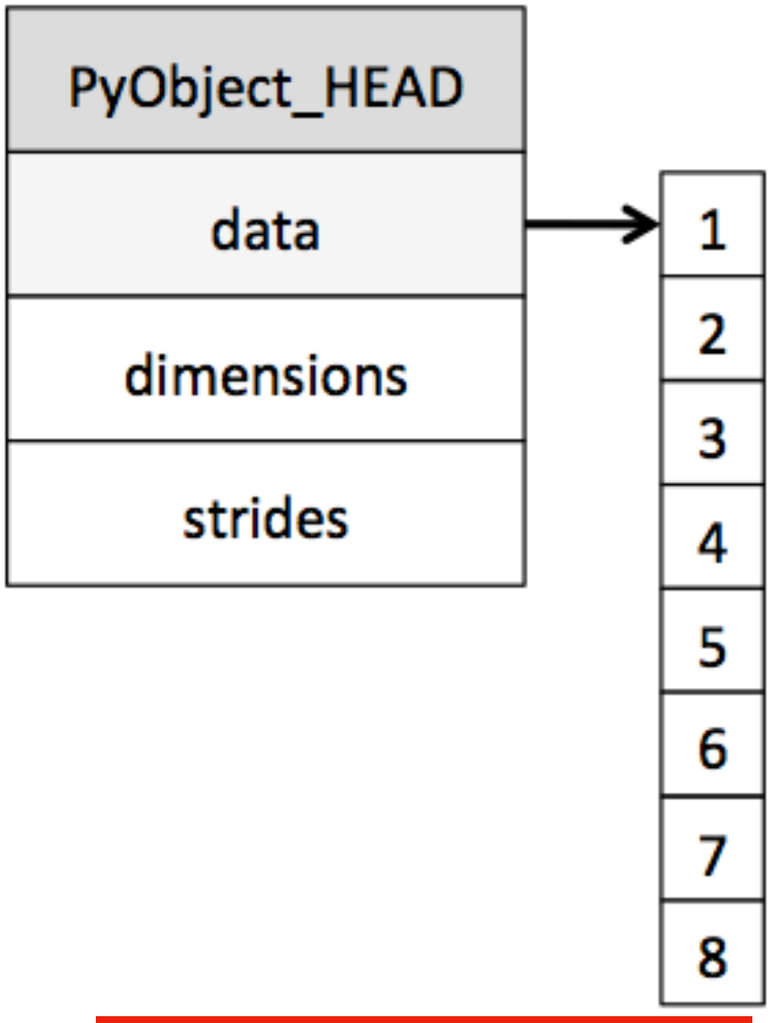
```
성능차이 34841.64192017862 배
```

numpy의 ndarray 객체
- 동일 자료형의 객체를
- 이웃한 메모리 위치에 삽입
- C언어의 배열과 유사하다

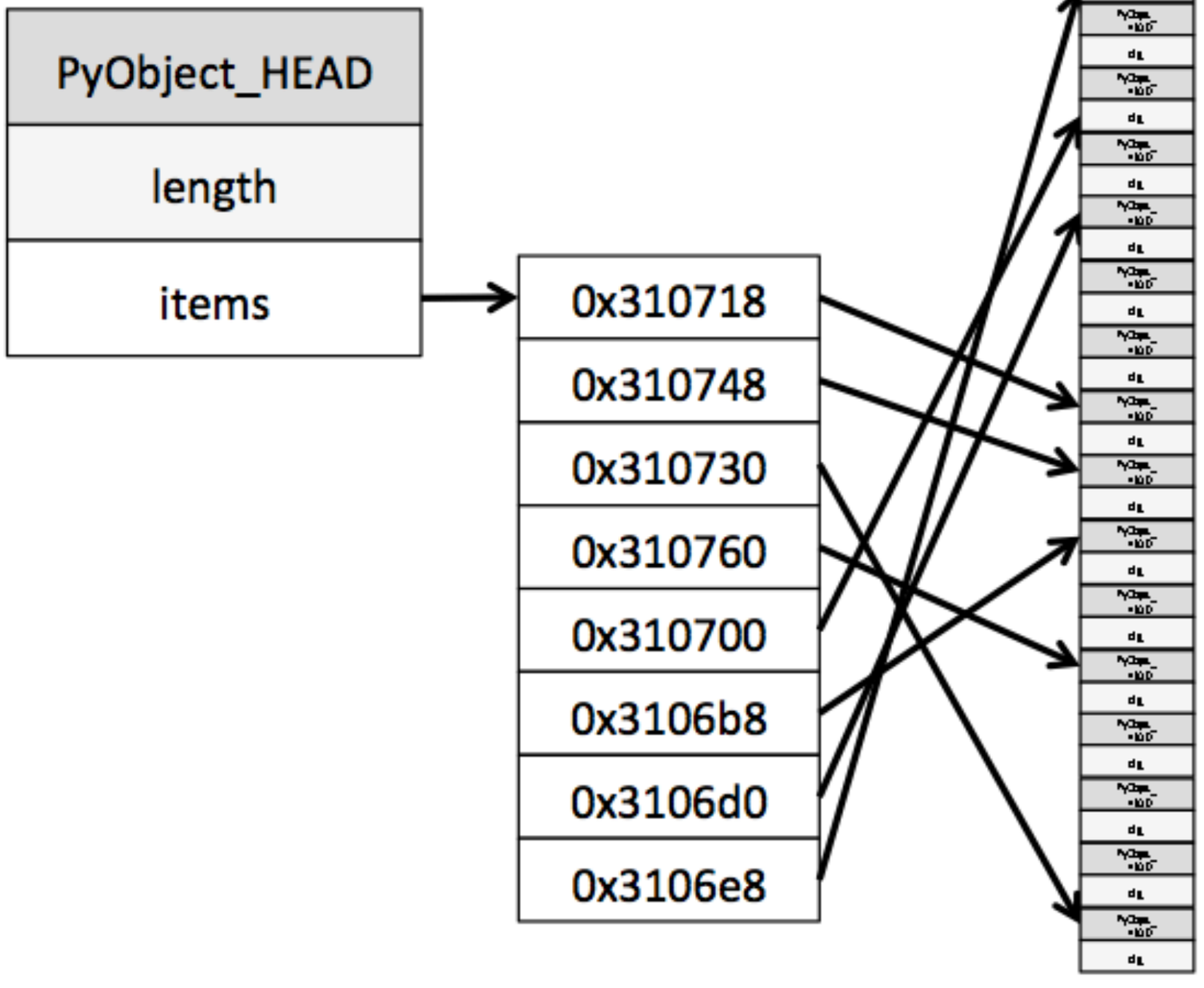


https://anaconda.org/arvindpdmn/numpy-basics/notebook?version=2018.10.13.1922

Numpy Array



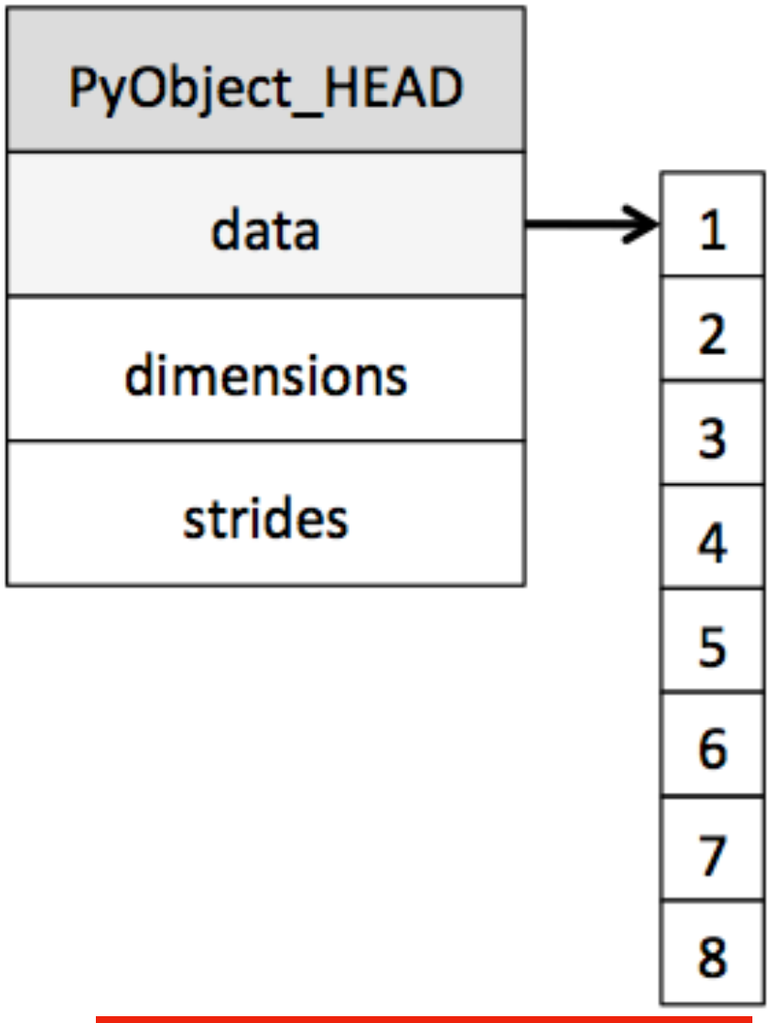
Python List



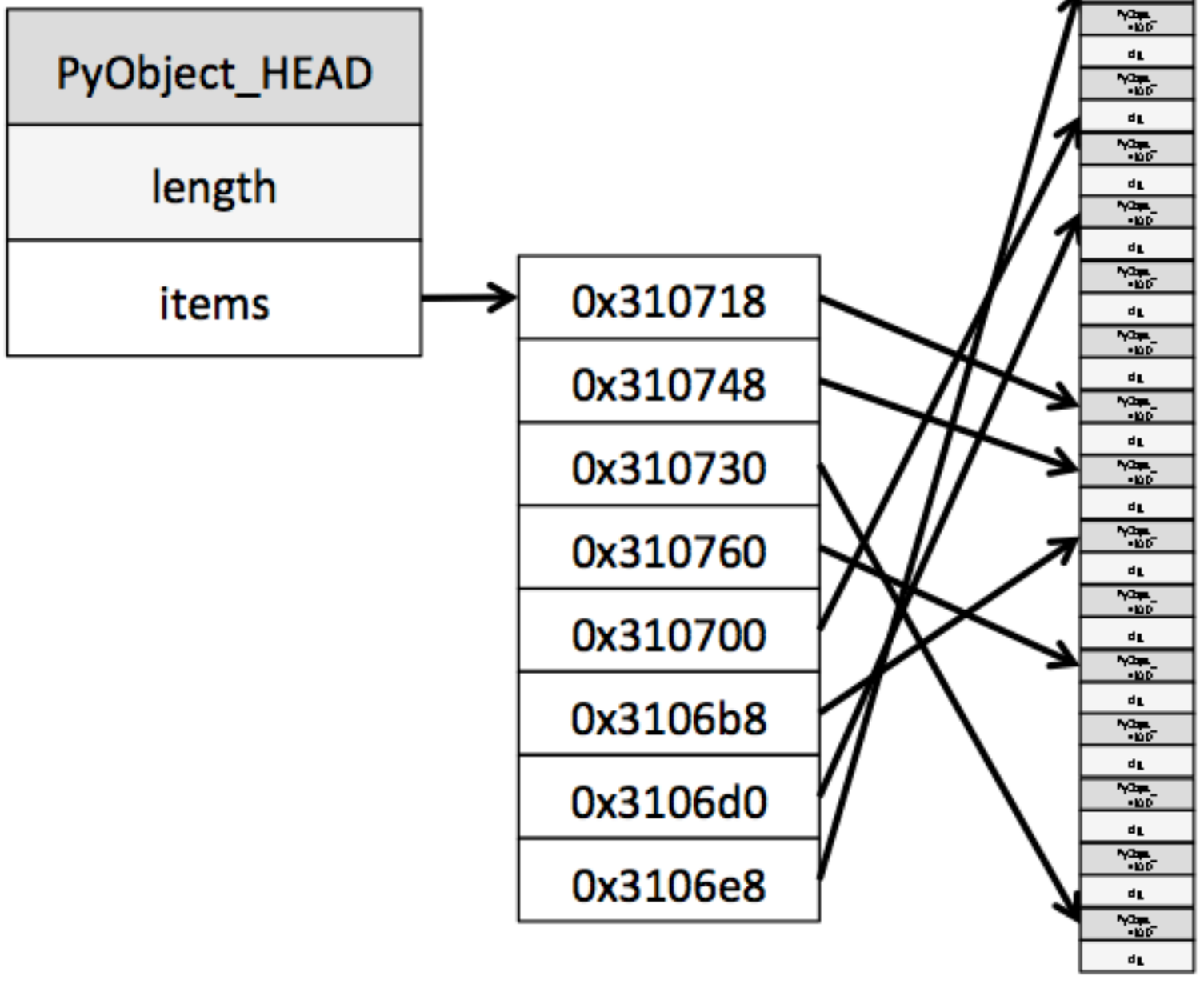
numpy의 ndarray 객체
- 동일 자료형의 객체를
- 이웃한 메모리 위치에 삽입
- C언어의 배열과 유사하다

https://anaconda.org/arvindpdmn/numpy-basics/notebook?version=2018.10.13.1922

Numpy Array



Python List



numpy의 ndarray 객체
- 동일 자료형의 객체를
- 이웃한 메모리 위치에 삽입
- C언어의 배열과 유사하다

https://anaconda.org/arvindpdmn/numpy-basics/notebook?version=2018.10.13.1922

정리

- 리바인딩은 시간이 많이 소요되더라
- 그래서 성능이 정말로 안 좋더라.
    - 파이썬은 느리다고 불평하는 경우가 이 경우이다
- 하지만 numpy를 사용하면 C와 비슷한 속도를 얻을 수 있다

라이브러리 최적화

- 파이썬의 동적할당 자체는 느리지만 최적화된 라이브러리들이 많다
- 파이썬은 느리다?
  - 그렇지 않다!!
  - 최적화된 라이브러리를 사용하면 좋은 성능을 보인다

감사합니다.