

AI 공부를 위한 준비! Deep into the 파이썬

창원대학교 정보통신공학과 교수 박동규

AI 공부를 위한 준비! Deep into the 파이썬

창원대학교 정보통신공학과 교수 박동규

Welcome to Busan.

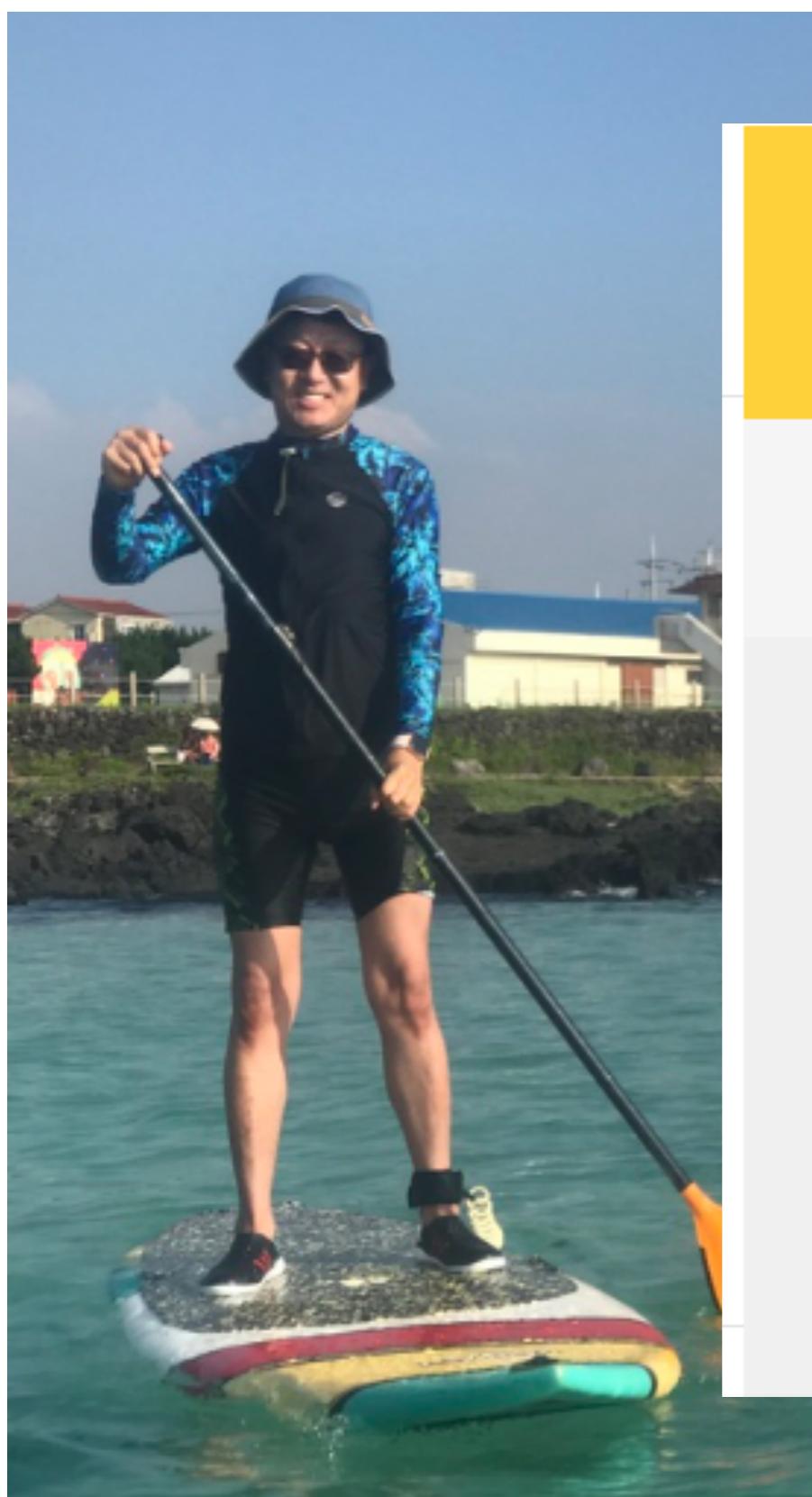


강사소개

강사소개



강사소개



A screenshot of a YouTube channel page. At the top, there is a large yellow header with the channel name '날날한 교수의 Coding Class' in bold black and red text. To the left of the text is a blue icon containing a white play button. Below the header, there is a small profile picture of a man with glasses and a blue shirt, followed by the channel name 'Donggyu Park' and the subscriber count '구독자 1.02천명'. To the right of the channel name are two blue buttons: '채널 링크설정' and 'YOUTUBE 스튜디오'. Below the header, there is a navigation bar with tabs: '음' (selected), '동영상' (highlighted with a blue underline), '재생목록', '커뮤니티', '채널', and '정보'. To the right of the navigation bar is a search icon. The main content area shows a grid of video thumbnails. Each thumbnail includes the video title, view count, and upload date. The thumbnails are arranged in three rows. The first row contains five thumbnails: '02_3 합집과 튜플 축약' (조회수 27회, 1주 전), '02_2 리스트 축약 표현2' (조회수 58회, 2주 전), '02_1 리스트 축약 표현1' (조회수 50회, 3주 전), '01_5_동적 타이핑과 정적 타이핑' (조회수 35회, 3주 전), and '2018년 5월 네팔 방문 vlog 영상' (조회수 48회, 3개월 전). The second row contains five thumbnails: '01_4 다차원 리스트의 참조' (조회수 110회, 6개월 전), '01_3 합집, 많은 복사와 같은 복사' (조회수 176회, 6개월 전), '01-2 리스트 오소는 참조형이다' (조회수 89회, 6개월 전), '01-1 파이썬 객체와 참조 변수' (조회수 357회, 6개월 전), and '2019 상원시 스마트 맵센터 iOS 교육생 모집' (조회수 157회, 9개월 전). The third row contains four thumbnails: '파이썬' (partially visible), '들어쓰기' (partially visible), '선형 연립방정식 풀이' (조회수 3회, 9개월 전), and '3차원 배열의 인덱스' (partially visible). On the far right edge of the page, there is a vertical scroll bar.

강의자료

<https://github.com/dongupak/AI-Boot-Camp-Busan-2019>

AI, AI, AI!!

- 정말로 중요한 기술입니다.
- 미래의 먹거리입니다.
- 더 중요한 것은 기본이라고 생각합니다.
 - 프로그래밍, 자료구조, 알고리즘, 수학적 지식
 - 컴퓨터를 이용한 문제해결 능력

AI HAS A LONG HISTORY OF BEING “THE NEXT BIG THING”...

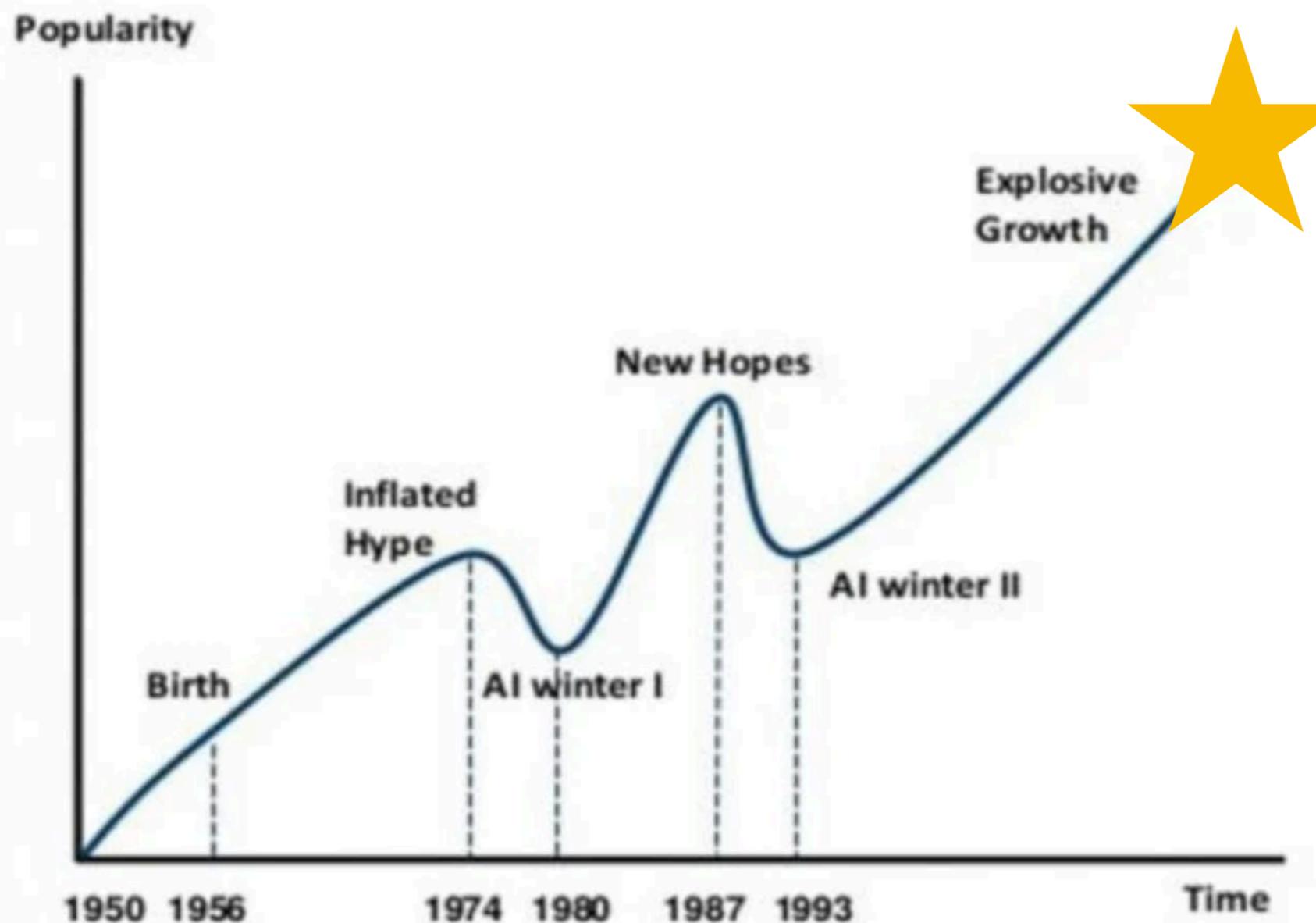
Popularity



Timeline of AI Development

- **1950s-1960s:** First AI boom - the age of reasoning, prototype AI developed
- **1970s:** AI winter I
- **1980s-1990s:** Second AI boom: the age of Knowledge representation (appearance of expert systems capable of reproducing human decision-making)
- **1990s:** AI winter II
- **1997:** Deep Blue beats Gary Kasparov
- **2006:** University of Toronto develops Deep Learning
- **2011:** IBM's Watson won Jeopardy
- **2016:** Go software based on Deep Learning beats world's champions

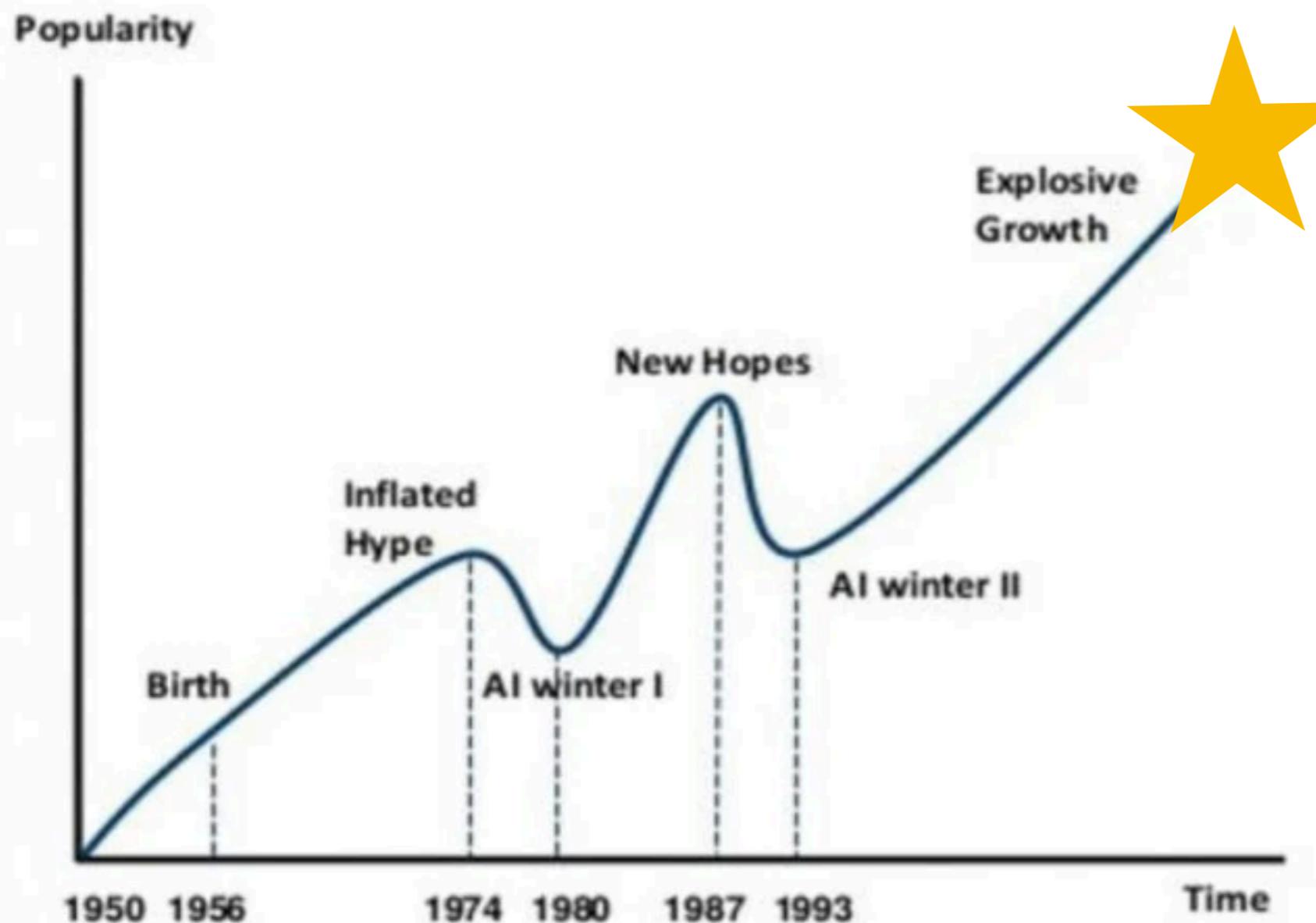
AI HAS A LONG HISTORY OF BEING “THE NEXT BIG THING”...



Timeline of AI Development

- **1950s-1960s**: First AI boom - the age of reasoning, prototype AI developed
- **1970s**: AI winter I
- **1980s-1990s**: Second AI boom: the age of Knowledge representation (appearance of expert systems capable of reproducing human decision-making)
- **1990s**: AI winter II
- **1997**: Deep Blue beats Gary Kasparov
- **2006**: University of Toronto develops Deep Learning
- **2011**: IBM's Watson won Jeopardy
- **2016**: Go software based on Deep Learning beats world's champions

AI HAS A LONG HISTORY OF BEING “THE NEXT BIG THING”...



Timeline of AI Development

- **1950s-1960s**: First AI boom - the age of reasoning, prototype AI developed
- **1970s**: AI winter I
- **1980s-1990s**: Second AI boom: the age of Knowledge representation (appearance of expert systems capable of reproducing human decision-making)
- **1990s**: AI winter II
- **1997**: Deep Blue beats Gary Kasparov
- **2006**: University of Toronto develops Deep Learning
- **2011**: IBM's Watson won Jeopardy
- **2016**: Go software based on Deep Learning beats world's champions

파이썬

파이썬의 성장세

Nov 2019	Nov 2018	Change	Programming Language	Ratings	Change
1	1		Java	16.246%	-0.50%
2	2		C	16.037%	+1.64%
3	4	▲	Python	9.842%	+2.16%
4	3	▼	C++	5.605%	-2.68%
5	6	▲	C#	4.316%	+0.36%
6	5	▼	Visual Basic .NET	4.229%	-2.26%
7	7		JavaScript	1.929%	-0.73%
8	8		PHP	1.720%	-0.66%

파이썬의 성장세

Nov 2019	Nov 2018	Change	Programming Language	Ratings	Change
1			Python popularity reaches an all-time high		-0.50%
2			At the current pace of growth, Tiobe estimates Python could surpass Java and C in popularity in three to four years		+1.64%
3					+2.16%
4			 		-2.68%
5			By Paul Krill Editor at Large, InfoWorld JUN 10, 2019		+0.36%
6					-2.26%
7					-0.73%
8					-0.66%



파이썬의 성장세

Nov 2019	Nov 2018	Change	Programming Language	Ratings	Change
1			Python popularity reaches an all-time high		-0.50%
2			At the current pace of growth, Tiobe estimates Python could surpass Java and C in popularity in three to four years		+1.64%
3					+2.16%
4					-2.68%
5	 By Paul Krill Editor at Large, I		Python has reached its highest rating ever in the monthly Tiobe index of programming language popularity. On its current trajectory, Tiobe noted, Python could leapfrog Java and C in the next three or four years to become the index's most-popular language.		
6					
7					
8				The June Tiobe rating of 8.53 percent for Python tops its previous high of 8.376 percent achieved last December . Python remains in third place behind Java and C. Python offers an ease of use that Java and C do not and is attracting a lot of newcomers, Tiobe reasoned.	

[[What is Python? Everything you need to know.](#) • [Tutorial: How to get started with Python.](#) • [6 essential libraries for every Python developer.](#) • [Why you should use Python for machine learning.](#) | [Keep up with hot topics in programming with InfoWorld's App Dev Report newsletter.](#)]

파이썬의 성장세

Nov 2019	Nov 2018	Change	Programming Language	Ratings	Change
1			Python popularity reaches an all-time high		-0.50%
2			At the current pace of growth, Tiobe estimates Python could surpass Java and C.		+1.64%
3			현재의 성장세를 이어간다면		
4			파이썬은 향후 3-4년 이내 가장 인기있는 프로그래밍		
5			언어로 자리매김할 전망		
6			(TIOBE Index의 1등이 수십년 만에 바뀔 수 있다)		
7					
8					

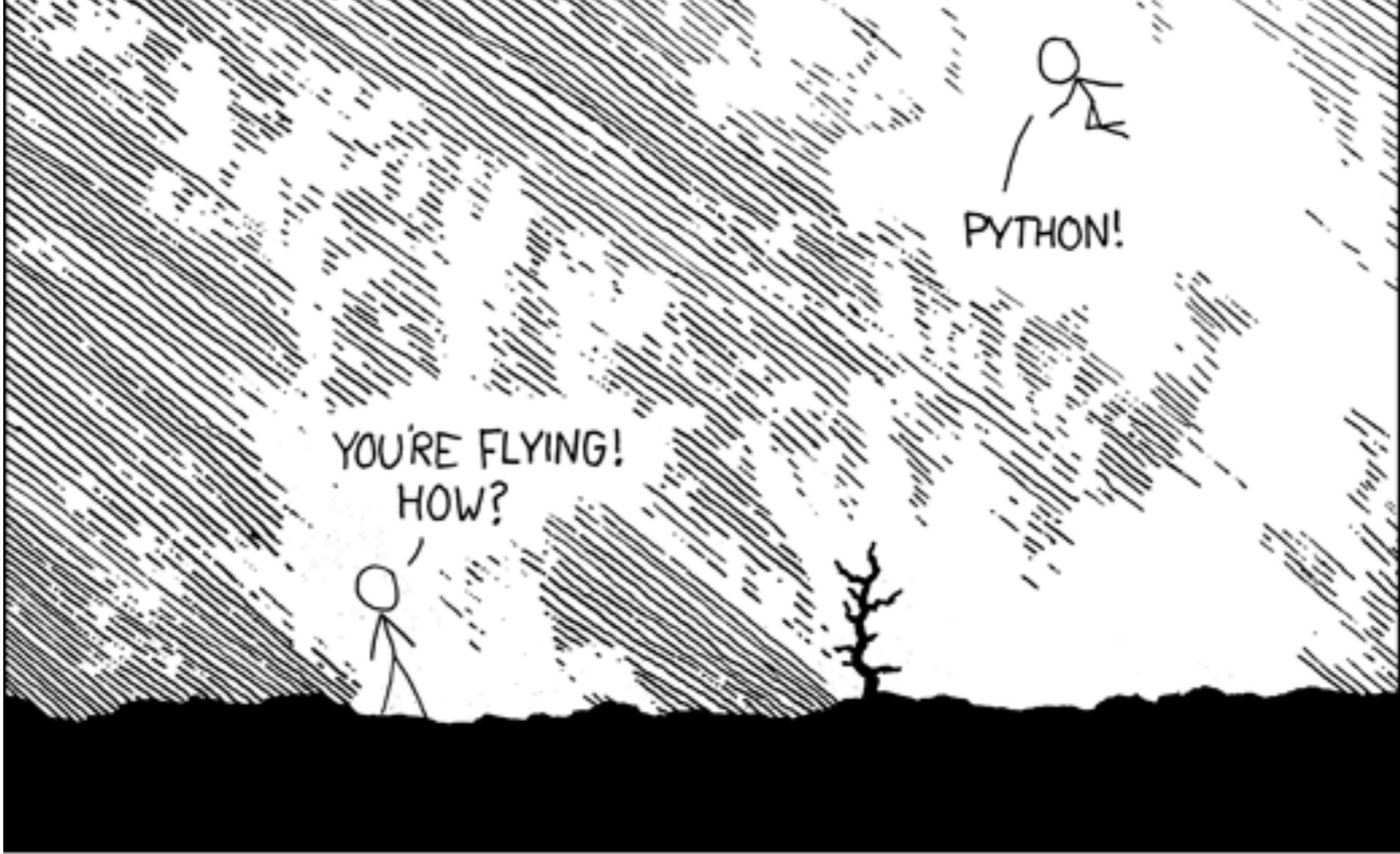


The June Tiobe rating of 8.55 percent for Python tops its previous high of 8.376 percent achieved last December. Python remains in third place behind Java and C. Python offers an ease of use that Java and C do not and is attracting a lot of newcomers, Tiobe reasoned.

[[What is Python? Everything you need to know.](#) • [Tutorial: How to get started with Python.](#) • [6 essential libraries for every Python developer.](#) • [Why you should use Python for machine learning.](#) | [Keep up with hot topics in programming with InfoWorld's App Dev Report newsletter.](#)]

파이썬의 장점

- 외부 모듈이 엄청 많음
- 리이브러리도 빵빵하더라
- 오픈소스 방식으로 개발되고 커뮤니티도 활발하더라
 - 커뮤니티에서 주도적으로 개발하더라
- 배우기 쉽고 읽기 쉬운 코드를 만들 수 있다
- 리스트, 딕셔너리, 집합등 사용자 친화적인 자료구조
- 그래서.. 생산성이 엄청 높고 빨리 결과를 볼 수 있더라



I LEARNED IT LAST NIGHT! EVERYTHING IS SO SIMPLE!

HELLO WORLD IS JUST

`print "Hello, world!"`

I DUNNO...
DYNAMIC TYPING?
WHITESPACE?

COME JOIN US!
PROGRAMMING
IS FUN AGAIN!
IT'S A WHOLE
NEW WORLD
UP HERE!

BUT HOW ARE
YOU FLYING?

I JUST TYPED
`import antigravity`

THAT'S IT?

... I ALSO SAMPLED
EVERYTHING IN THE
MEDICINE CABINET
FOR COMPARISON.

BUT I THINK THIS
IS THE PYTHON.

파이썬의 장점

- 머신러닝 기술의 구현에 가장 적합한 프로그래밍 언어

Python Libraries	Commits	Contributors	Star	Fork
Tensor Flow Python	61,100	2,095	131,727	76,381
Keras Python	5,143	805	43,011	16,390
Theano Python	28,081	330	8,854	2,495
Scikit-learn Python	24,286	1,369	36,333	17,819
PyTorch Python	19,513	1,114	30,284	7,393
NumPy Python	20,932	797	11,331	3,734
Python Pandas	19,846	1,534	20,542	8,124
Seaborn Python	2,316	98	6,287	1,031

단점

- 속도
 - C/C++ 보다 느리고 심지어 GO같은 언어보다 훨씬 느리다는 비판
- 모바일 컴퓨팅
 - 파이썬으로 만든 모바일 앱 보신분?
- 설계 제약
 - 동적 언어이다 보니 실행시간에만 오류가 드러나기도 하더라

Deep into the 파이썬

자주 이용하지만 남에게 설
명할 만큼 잘 알지는 못하는
핵심 개념~~~

Everything is an Object

Everything is an Object

- 파이썬은 객체지향 프로그래밍 언어

Everything is an Object

- 파이썬은 객체지향 프로그래밍 언어
- 파이썬은 객체가 중심이 되며, 참조 변수를 통해 객체에 접근.

Everything is an Object

- 파이썬은 객체지향 프로그래밍 언어
- 파이썬은 객체가 중심이 되며, 참조 변수를 통해 객체에 접근.
- C 언어는 변수가 생성되고 변수에 값이 저장되는 구조.

Everything is an Object

- 파이썬은 객체지향 프로그래밍 언어
- 파이썬은 객체가 중심이 되며, 참조 변수를 통해 객체에 접근.
- C 언어는 변수가 생성되고 변수에 값이 저장되는 구조.
- 파이썬의 변수는 동적으로 참조하는 객체가 지정됨.

자료형

- 파이썬은 정수형, 실수형, 복소수형, 문자열, 리스트 등의 다양한 자료형이 있다.
- 자료형에 따라서 지원하는 연산자가 다르며 메소드들도 다르다.
- 사용자의 필요성에 따라 적절한 자료형을 선택하여 사용한다.

자료형

형식

설명

자료형	형식	설명
int 형	$n = 100$ $n += 200$	음의 정수, 0, 양의 정수 값을 가지며 사칙연산자를 비롯한 연산자를 사용할 수 있다

자료형	형식	설명
int 형	$n = 100$ $n += 200$	음의 정수, 0, 양의 정수 값을 가지며 사칙연산자를 비롯한 연산자를 사용할 수 있다
float 형	$f = 12.3$ $f = f - 30.12$	소숫점 아래 숫자를 가질 수 있으며 정밀한 숫자 표현이 가능하다. 정수형에서 사용하는 연산자를 많이 사용할 수 있다.

자료형	형식	설명
int 형	$n = 100$ $n += 200$	음의 정수, 0, 양의 정수 값을 가지며 사칙연산자를 비롯한 연산자를 사용할 수 있다
float 형	$f = 12.3$ $f = f - 30.12$	소숫점 아래 숫자를 가질 수 있으며 정밀한 숫자 표현이 가능하다. 정수형에서 사용하는 연산자를 많이 사용할 수 있다.
complex 형	$a = 3 + 2.0J$	실수부와 허수부를 가지는 복소수를 표현할 수 있는 자료형

자료형	형식	설명
int 형	$n = 100$ $n += 200$	음의 정수, 0, 양의 정수 값을 가지며 사칙연산자를 비롯한 연산자를 사용할 수 있다
float 형	$f = 12.3$ $f = f - 30.12$	소숫점 아래 숫자를 가질 수 있으며 정밀한 숫자 표현이 가능하다. 정수형에서 사용하는 연산자를 많이 사용할 수 있다.
complex 형	$a = 3 + 2.0J$	실수부와 허수부를 가지는 복소수를 표현할 수 있는 자료형
str 형	$s = "hello"$ $s = s * 3$ $s.upper()$	문자열을 저장함 문자열 반복 연산자를 지원 $upper()$, $lower()$, $split()$ 등 메소드 지원

정수 객체와 변수

- 정수형 객체가 있으면 이 객체에 연산자를 적용하여 연산을 수행할 수 있다.
- 변수에 데이터를 보관하고 필요할 때 참조하면 편리하다

```
>>> a = 100
>>> print(a * 10)
1000
>>> print(a * 20)
2000
>>> a = 200
>>> print(a * 10)
2000
>>> print(a * 20)
4000
```

C

상자 모형

Python

꼬리표(태그) 모형

C

상자 모형

```
int a;      // 정수형 변수 선언  
a = 100;
```

Python

꼬리표(태그) 모형

C

상자 모형

```
int a;      // 정수형 변수 선언  
a = 100;
```

Python

꼬리표(태그) 모형

a는 메모리에 있는 100값을
담는 상자

C

상자 모형

```
int a; // 정수형 변수 선언  
a = 100;
```

Python

꼬리표(태그) 모형

a :



a는 메모리에 있는 100값을
담는 상자

C

상자 모형

```
int a; // 정수형 변수 선언  
a = 100;
```

a : 100

a는 메모리에 있는 100값을
담는 상자

Python

꼬리표(태그) 모형

C

상자 모형

```
int a; // 정수형 변수 선언  
a = 100;
```

a : 100

a는 메모리에 있는 100값을
담는 상자

Python

꼬리표(태그) 모형

C

상자 모형

```
int a; // 정수형 변수 선언  
a = 100;
```

a : 100

a는 메모리에 있는 100값을
담는 상자

Python

꼬리표(태그) 모형

```
a = 100 # 객체 생성
```

C

상자 모형

```
int a; // 정수형 변수 선언  
a = 100;
```

a : 100

a는 메모리에 있는 100값을
담는 상자

Python

꼬리표(태그) 모형

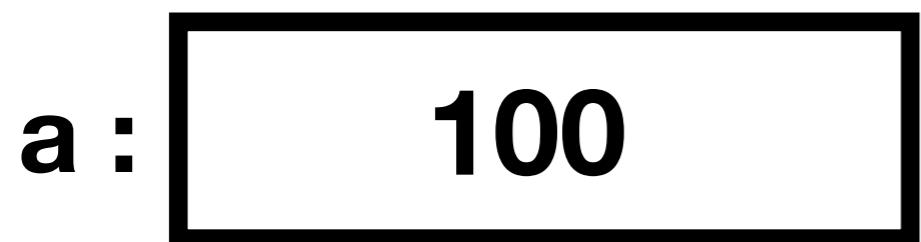
```
a = 100 # 객체 생성
```

100

C

상자 모형

```
int a; // 정수형 변수 선언  
a = 100;
```

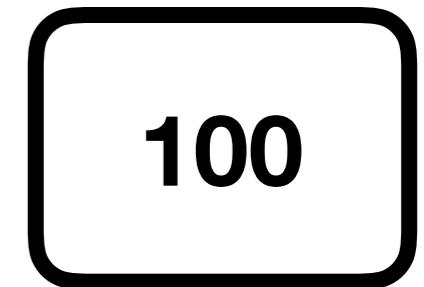
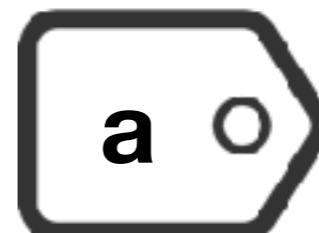


a는 메모리에 있는 100값을
담는 상자

Python

꼬리표(태그) 모형

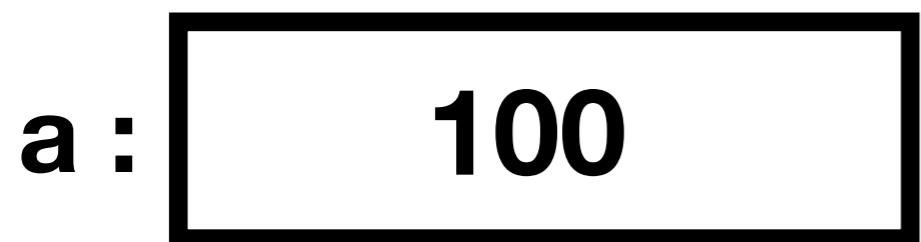
```
a = 100 # 객체 생성
```



C

상자 모형

```
int a; // 정수형 변수 선언  
a = 100;
```

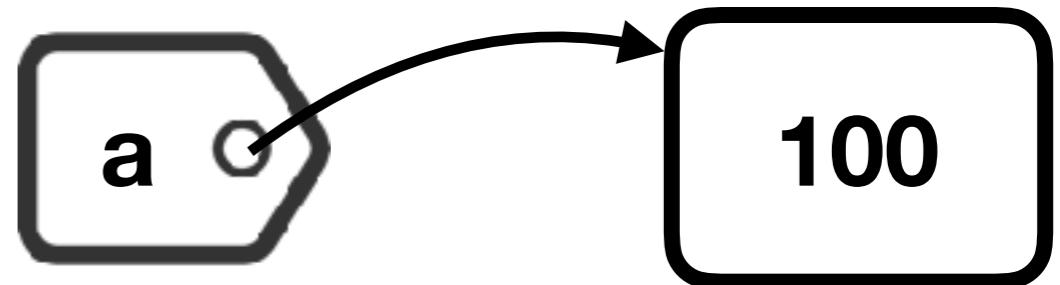


a는 메모리에 있는 100값을
담는 상자

Python

꼬리표(태그) 모형

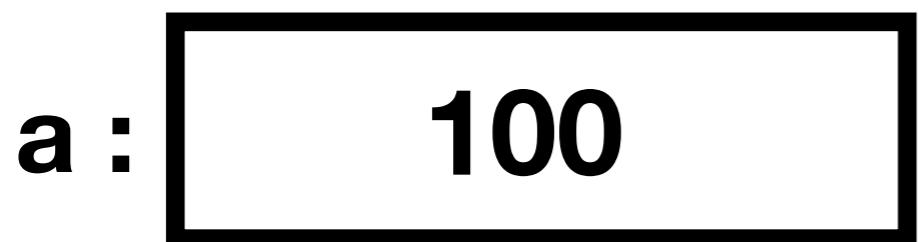
```
a = 100 # 객체 생성
```



C

상자 모형

```
int a; // 정수형 변수 선언  
a = 100;
```

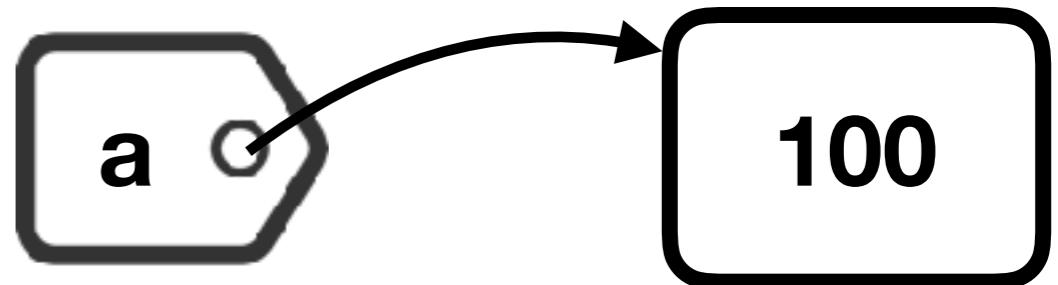


a는 메모리에 있는 100값을
담는 상자

Python

꼬리표(태그) 모형

```
a = 100 # 객체 생성
```

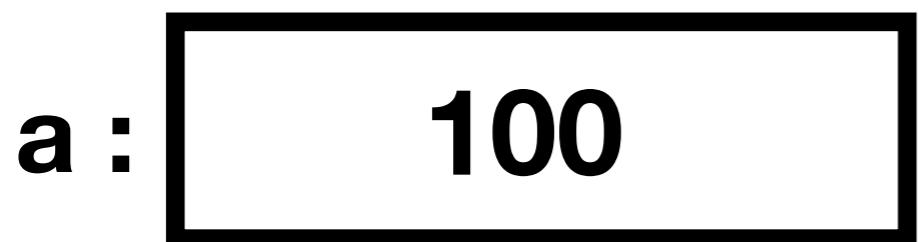


a는 객체에 대한 꼬리표(태그)

C

상자 모형

```
int a; // 정수형 변수 선언  
a = 100;
```

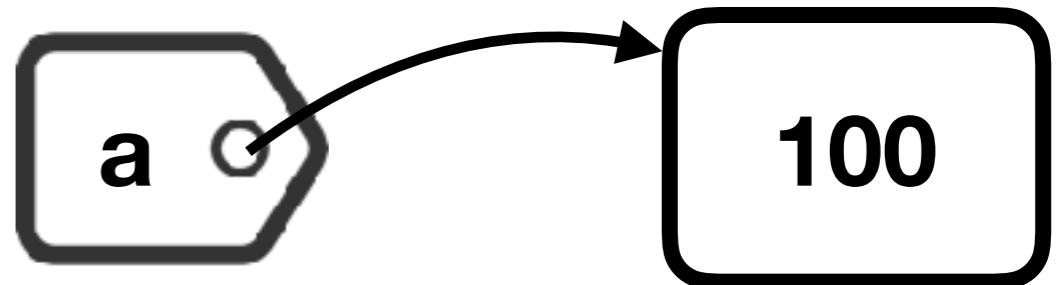


a는 메모리에 있는 100값을
담는 상자

Python

꼬리표(태그) 모형

```
a = 100 # 객체 생성
```



a는 객체에 대한 꼬리표(태그)

C

변수 중심

Python

객체 중심

C

변수 중심

```
int a; // 정수형 변수 선언
```

Python

객체 중심

C

변수 중심

int a; // 정수형 변수 선언

변수 선언과 동시에 메모리 할당

Python

객체 중심

C

변수 중심

```
int a; // 정수형 변수 선언
```

a : 

변수 선언과 동시에 메모리 할당

Python

객체 중심

C

변수 중심

```
int a; // 정수형 변수 선언
```

```
a = 100; // 정수값 할당
```

a : 

변수 선언과 동시에 메모리 할당

Python

객체 중심

C

변수 중심

```
int a; // 정수형 변수 선언
```

```
a = 100; // 정수값 할당
```

a : 100

변수 선언과 동시에 메모리 할당

Python

객체 중심

C

변수 중심

```
int a; // 정수형 변수 선언
```

```
a = 100; // 정수값 할당
```

```
a = 200; // 정수값 재할당
```

a : 100

변수 선언과 동시에 메모리 할당

Python

객체 중심

C

변수 중심

```
int a; // 정수형 변수 선언
```

```
a = 100; // 정수값 할당
```

```
a = 200; // 정수값 재할당
```

```
a : 
```

변수 선언과 동시에 메모리 할당

Python

객체 중심

C

변수 중심

```
int a; // 정수형 변수 선언
```

```
a = 100; // 정수값 할당
```

```
a = 200; // 정수값 재할당
```

a : 200

변수 선언과 동시에 메모리 할당

Python

객체 중심

C

변수 중심

```
int a; // 정수형 변수 선언
```

```
a = 100; // 정수값 할당
```

```
a = 200; // 정수값 재할당
```

a : 200

변수 선언과 동시에 메모리 할당

Python

객체 중심

C

변수 중심

```
int a; // 정수형 변수 선언
```

```
a = 100; // 정수값 할당
```

```
a = 200; // 정수값 재할당
```

a : 200

변수 선언과 동시에 메모리 할당

Python

객체 중심

```
a = 100 # 객체 생성
```

C

변수 중심

```
int a; // 정수형 변수 선언
```

```
a = 100; // 정수값 할당
```

```
a = 200; // 정수값 재할당
```

a : 200

변수 선언과 동시에 메모리 할당

Python

객체 중심

```
a = 100 # 객체 생성
```

100

C

변수 중심

```
int a; // 정수형 변수 선언
```

```
a = 100; // 정수값 할당
```

```
a = 200; // 정수값 재할당
```

a : 200

변수 선언과 동시에 메모리 할당

Python

객체 중심

```
a = 100 # 객체 생성  
# 객체에 대한 참조 변수 a
```

100

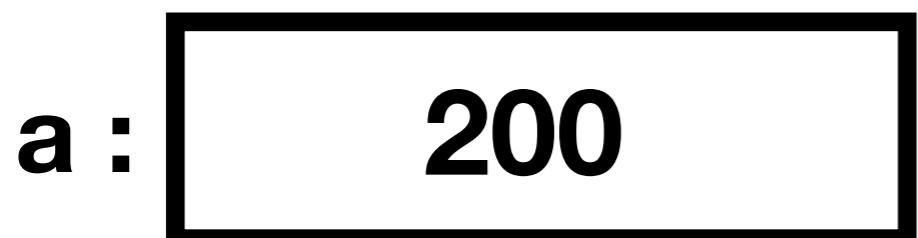
C

변수 중심

```
int a; // 정수형 변수 선언
```

```
a = 100; // 정수값 할당
```

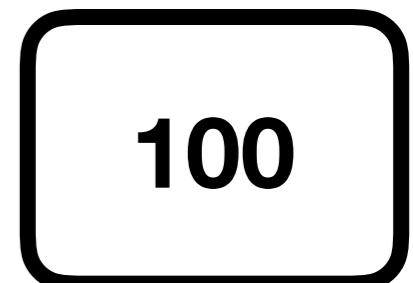
```
a = 200; // 정수값 재할당
```



Python

객체 중심

```
a = 100 # 객체 생성  
# 객체에 대한 참조 변수 a
```



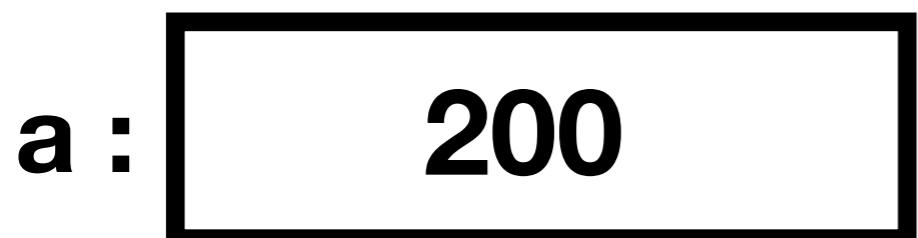
C

변수 중심

```
int a; // 정수형 변수 선언
```

```
a = 100; // 정수값 할당
```

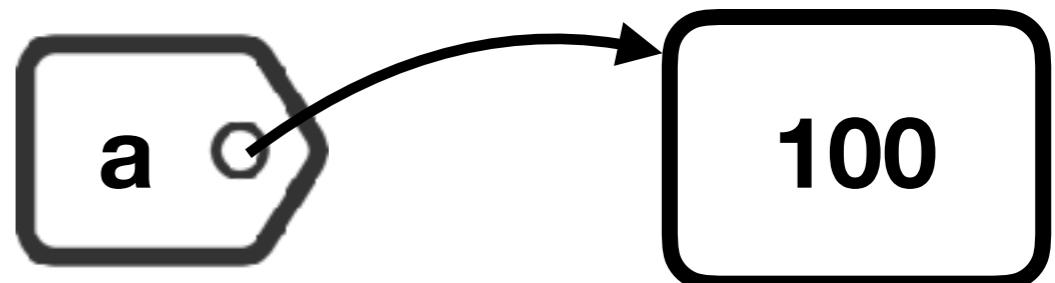
```
a = 200; // 정수값 재할당
```



Python

객체 중심

```
a = 100 # 객체 생성  
# 객체에 대한 참조 변수 a
```



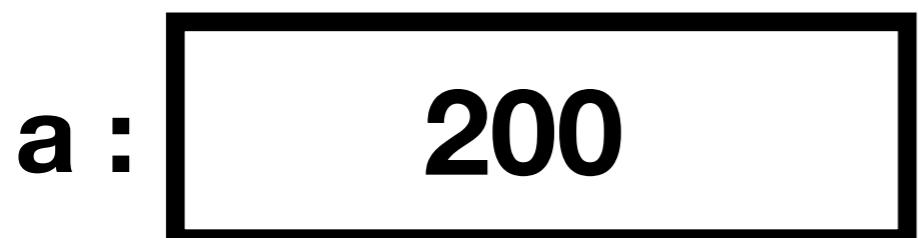
C

변수 중심

```
int a; // 정수형 변수 선언
```

```
a = 100; // 정수값 할당
```

```
a = 200; // 정수값 재할당
```



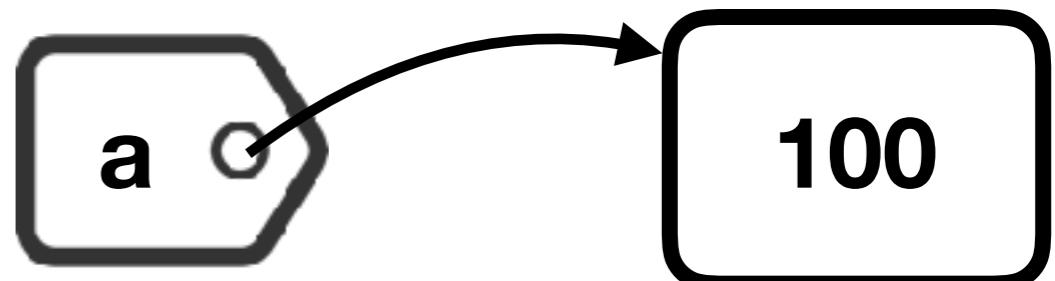
변수 선언과 동시에 메모리 할당

Python

객체 중심

```
a = 100 # 객체 생성  
# 객체에 대한 참조 변수 a
```

```
a = 200 # 객체 재할당
```



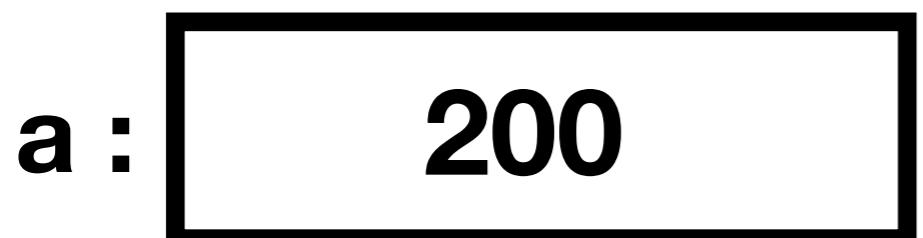
C

변수 중심

```
int a; // 정수형 변수 선언
```

```
a = 100; // 정수값 할당
```

```
a = 200; // 정수값 재할당
```



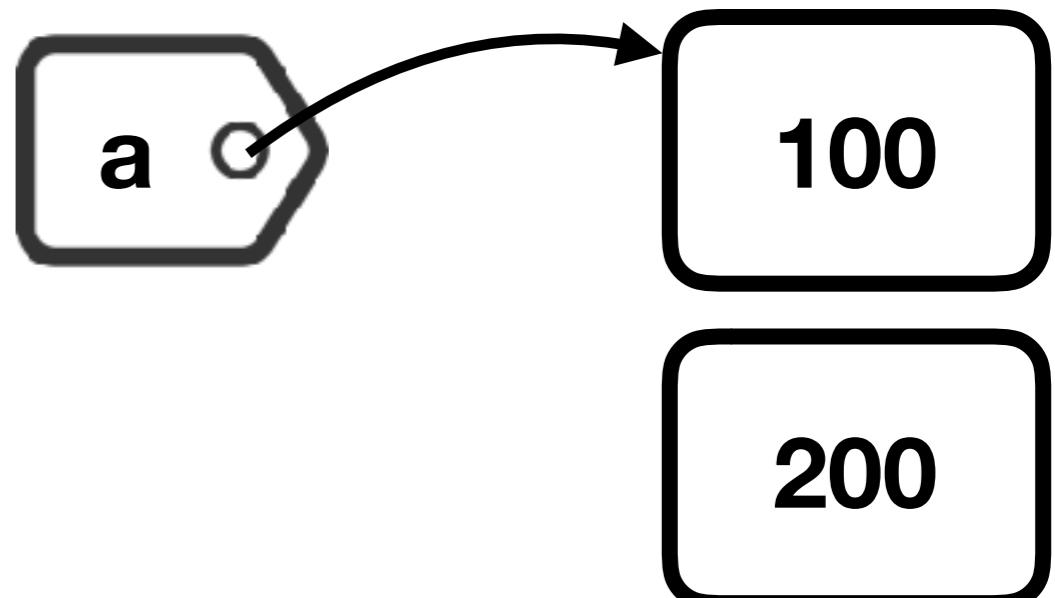
변수 선언과 동시에 메모리 할당

Python

객체 중심

```
a = 100 # 객체 생성  
# 객체에 대한 참조 변수 a
```

```
a = 200 # 객체 재할당
```



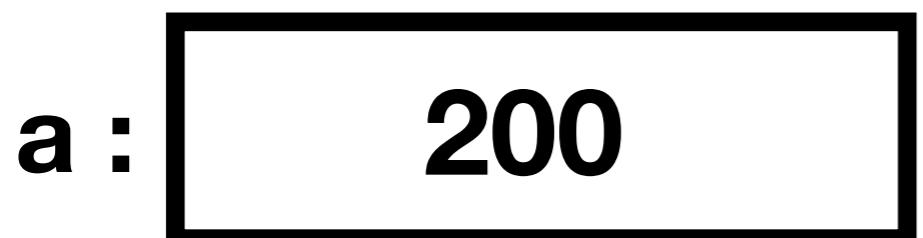
C

변수 중심

```
int a; // 정수형 변수 선언
```

```
a = 100; // 정수값 할당
```

```
a = 200; // 정수값 재할당
```

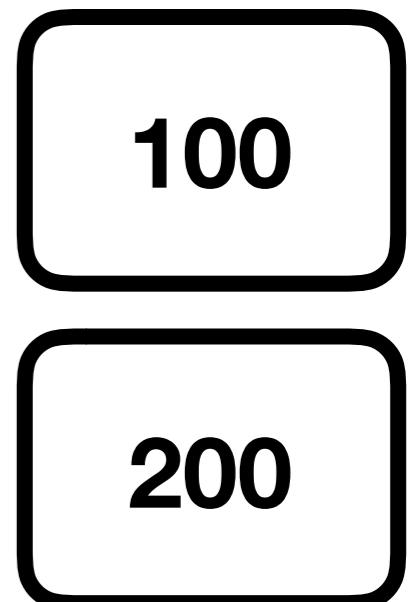


Python

객체 중심

```
a = 100 # 객체 생성  
# 객체에 대한 참조 변수 a
```

```
a = 200 # 객체 재할당
```



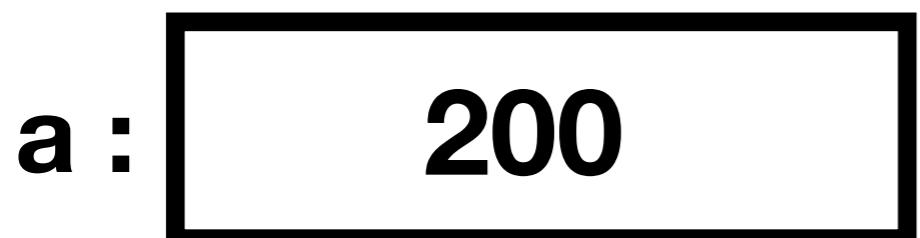
C

변수 중심

```
int a; // 정수형 변수 선언
```

```
a = 100; // 정수값 할당
```

```
a = 200; // 정수값 재할당
```

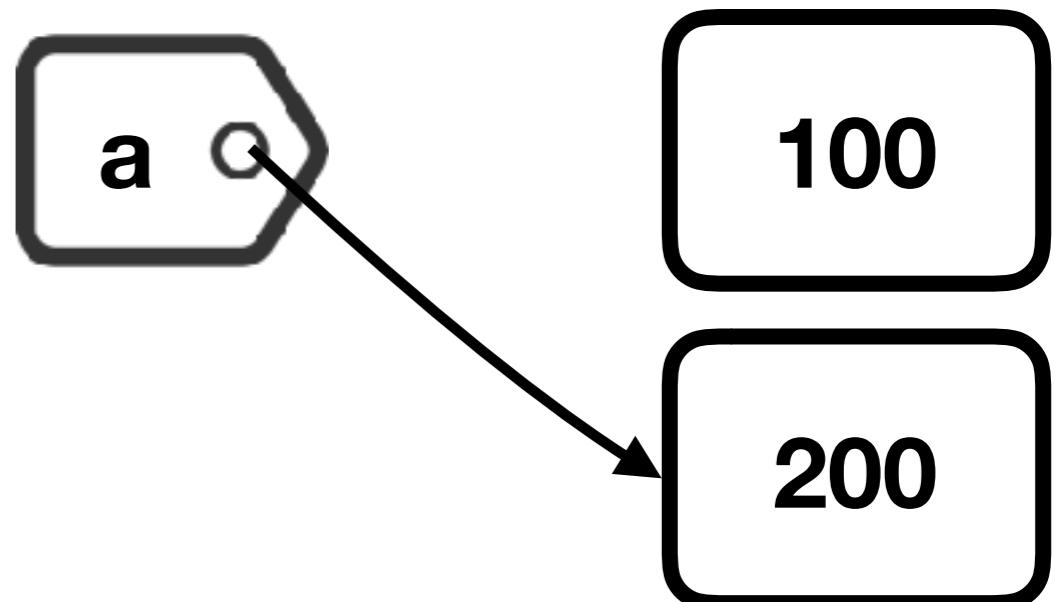


Python

객체 중심

```
a = 100 # 객체 생성  
# 객체에 대한 참조 변수 a
```

```
a = 200 # 객체 재할당
```



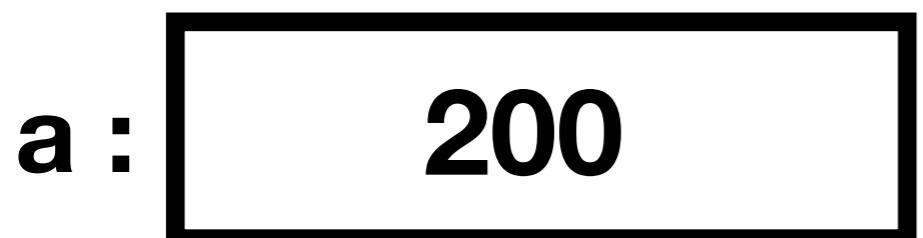
C

변수 중심

```
int a; // 정수형 변수 선언
```

```
a = 100; // 정수값 할당
```

```
a = 200; // 정수값 재할당
```

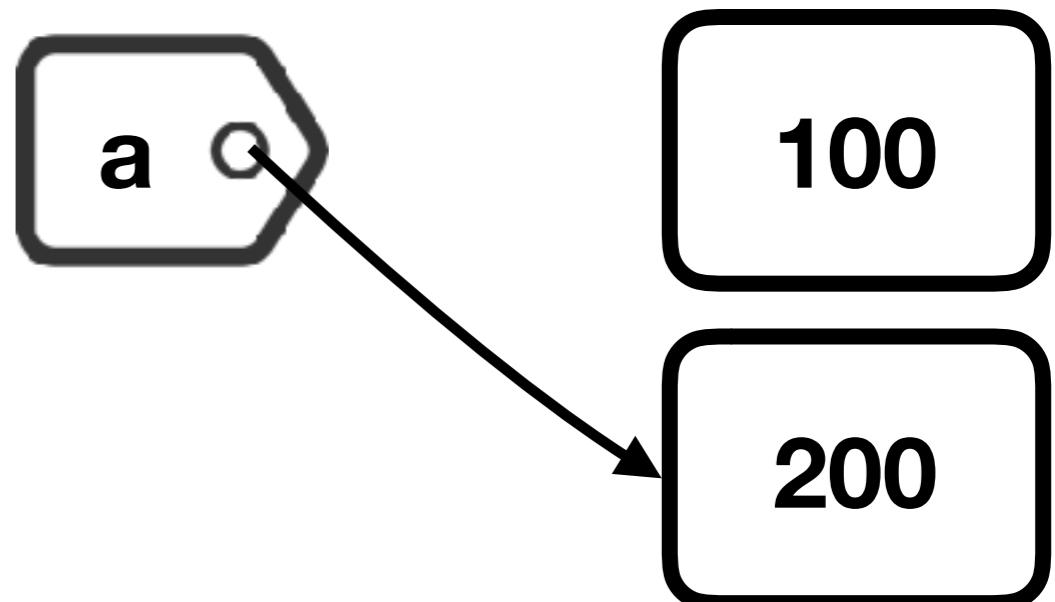


Python

객체 중심

```
a = 100 # 객체 생성  
# 객체에 대한 참조 변수 a
```

```
a = 200 # 객체 재할당
```



모든 객체는 고유한 id를 가진다

```
[>>> a = 100
[>>> print(id(a))
4324024528
[>>> print(id(100))
4324024528
```

모든 객체는 고유한 id를 가진다

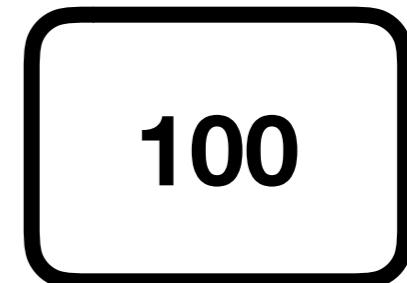
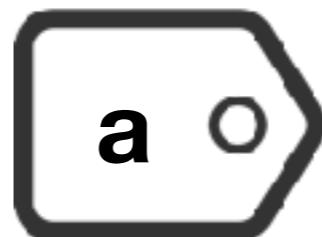
```
[>>> a = 100  
[>>> print(id(a))  
4324024528  
[>>> print(id(100))  
4324024528
```

100

id : 4324024528

모든 객체는 고유한 id를 가진다

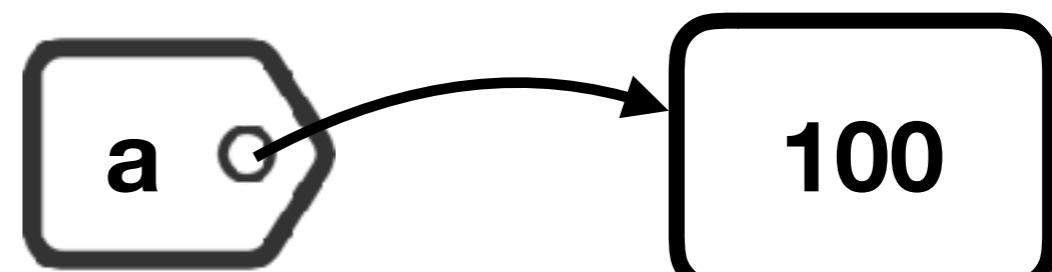
```
[>>> a = 100  
[>>> print(id(a))  
4324024528  
[>>> print(id(100))  
4324024528
```



id : 4324024528

모든 객체는 고유한 id를 가진다

```
[>>> a = 100  
[>>> print(id(a))  
4324024528  
[>>> print(id(100))  
4324024528
```

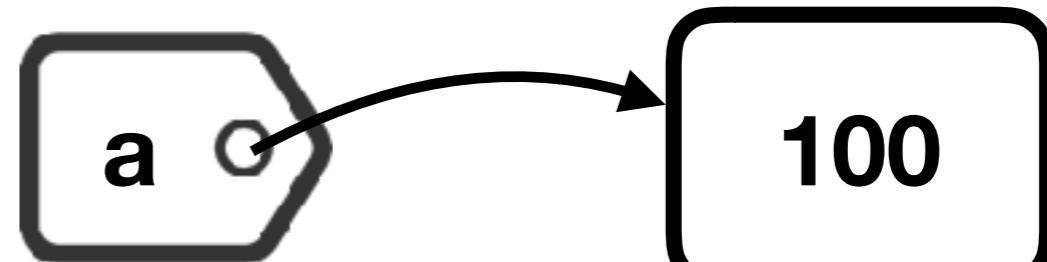


id : 4324024528

모든 객체는 고유한 id를 가진다

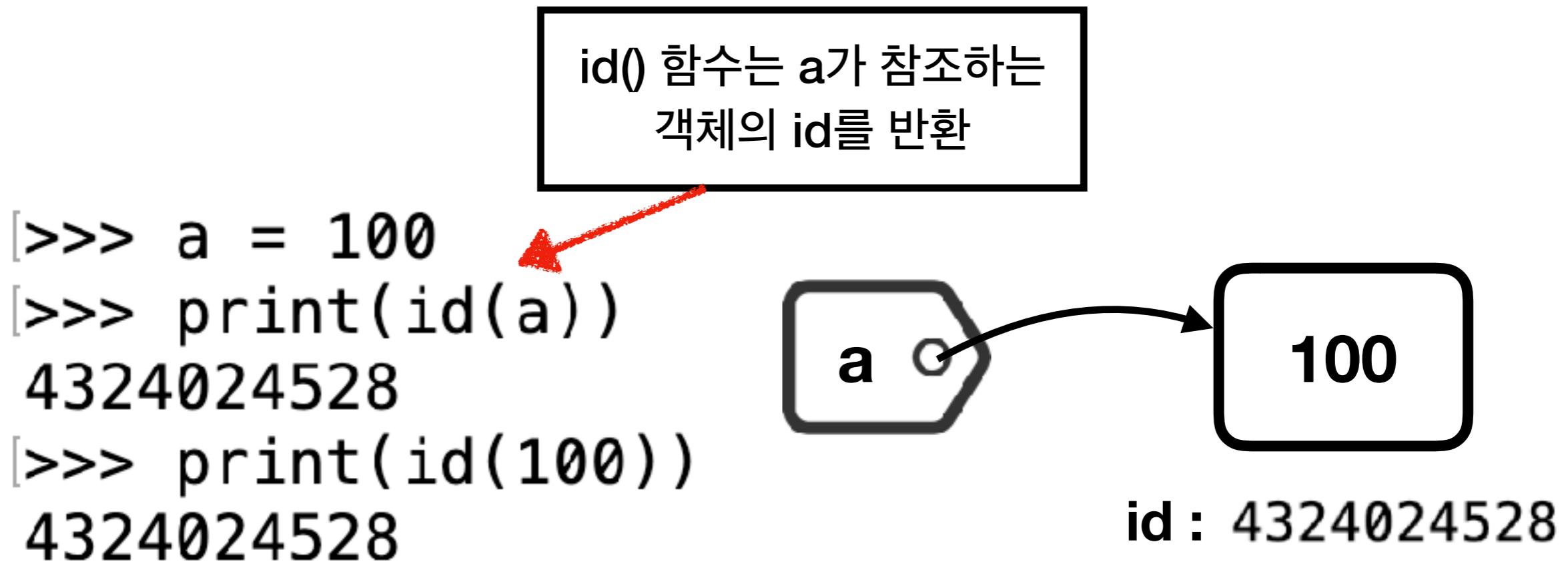
id() 함수는 a가 참조하는
객체의 id를 반환

```
[>>> a = 100
[>>> print(id(a))
4324024528
[>>> print(id(100))
4324024528
```



id : 4324024528

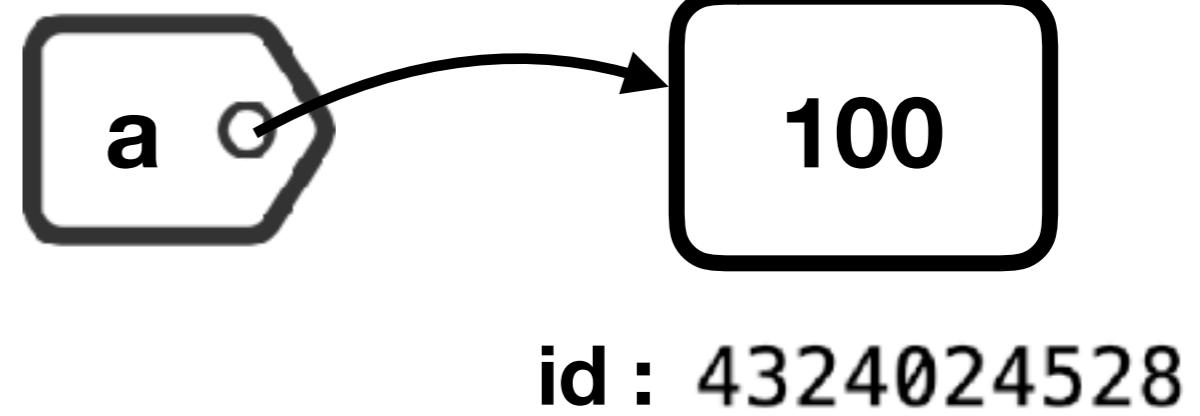
모든 객체는 고유한 id를 가진다



모든 객체는 고유한 id를 가진다

```
[>>> a = 100  
[>>> print(id(a))  
4324024528  
[>>> print(id(100))  
4324024528
```

id() 함수는 a가 참조하는
객체의 id를 반환

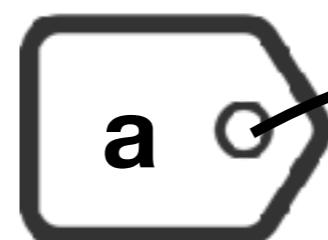


100 값을 가지는 정수
형 객체의 id

모든 객체는 고유한 id를 가진다

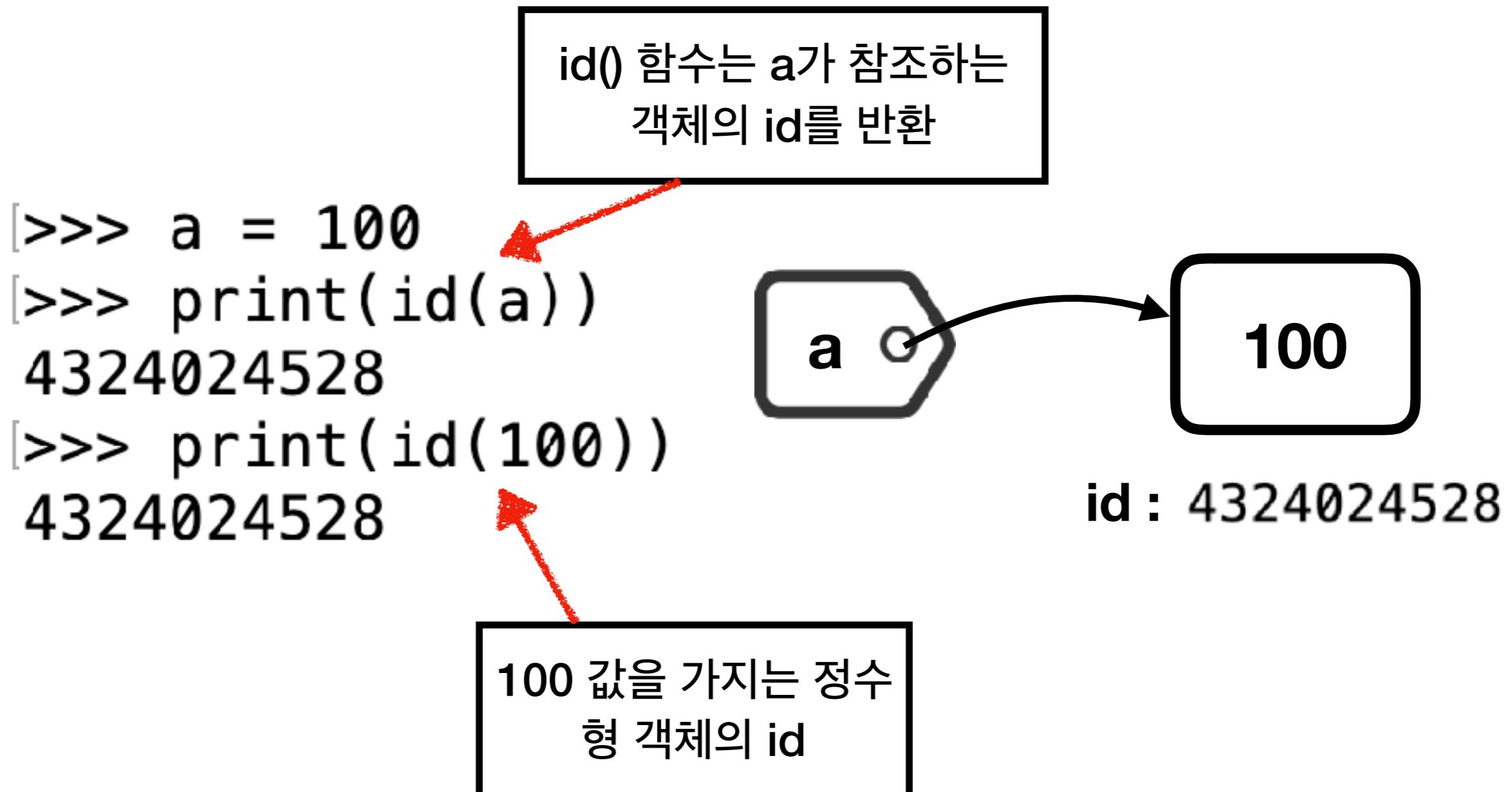
```
[>>> a = 100  
[>>> print(id(a))  
4324024528  
[>>> print(id(100))  
4324024528
```

id() 함수는 a가 참조하는
객체의 id를 반환



100 값을 가지는 정수
형 객체의 id

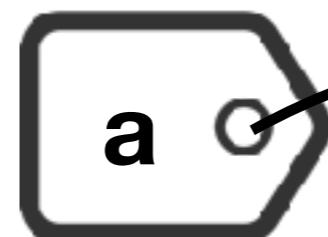
모든 객체는 고유한 id를 가진다



모든 객체는 고유한 id를 가진다

```
[>>> a = 100  
[>>> print(id(a))  
4324024528  
[>>> print(id(100))  
4324024528
```

id() 함수는 a가 참조하는
객체의 id를 반환



100 값을 가지는 정수
형 객체의 id

= 연산자가 하는 일

- 객체에 대한 참조를 만들거나 변경시킨다.
- 파이썬은 객체 중심적인 프로그래밍 언어이므로 객체와 객체를 참조하는 변수, 그리고 참조의 변경이라는 개념이 중요할 수 밖에 없다.
- 객체를 여러 변수가 동시에 참조할 수 있다.

객체를 여러 변수가 동시에 참조할 수 있다.

= (할당 연산자)는 객체에 대한 참조를 만들어 준다.

```
[>>> a = 100
[>>> b = a
[>>> print(id(a))
4324024528
[>>> print(id(b))
4324024528
```

객체를 여러 변수가 동시에 참조할 수 있다.

= (할당 연산자)는 객체에 대한 참조를 만들어 준다.

```
[>>> a = 100
```

100

```
[>>> b = a
```

```
[>>> print(id(a))
```

id : 4324024528

4324024528

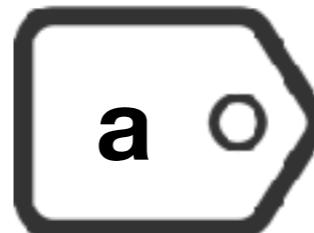
```
[>>> print(id(b))
```

4324024528

객체를 여러 변수가 동시에 참조할 수 있다.

= (할당 연산자)는 객체에 대한 참조를 만들어 준다.

```
[>>> a = 100
```



```
[>>> b = a
```

```
[>>> print(id(a))
```

4324024528

```
[>>> print(id(b))
```

4324024528

100

id : 4324024528

객체를 여러 변수가 동시에 참조할 수 있다.

= (할당 연산자)는 객체에 대한 참조를 만들어 준다.

```
[>>> a = 100
```

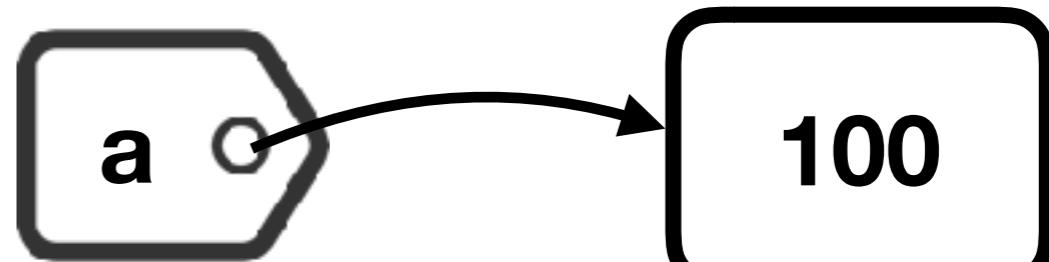
```
[>>> b = a
```

```
[>>> print(id(a))
```

```
4324024528
```

```
[>>> print(id(b))
```

```
4324024528
```



id : 4324024528

객체를 여러 변수가 동시에 참조할 수 있다.

= (할당 연산자)는 객체에 대한 참조를 만들어 준다.

```
[>>> a = 100
```

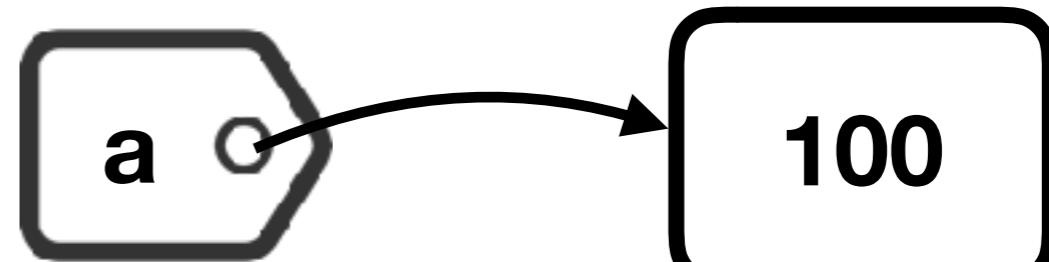
```
[>>> b = a
```

```
[>>> print(id(a))
```

```
4324024528
```

```
[>>> print(id(b))
```

```
4324024528
```



객체를 여러 변수가 동시에 참조할 수 있다.

= (할당 연산자)는 객체에 대한 참조를 만들어 준다.

```
[>>> a = 100
```

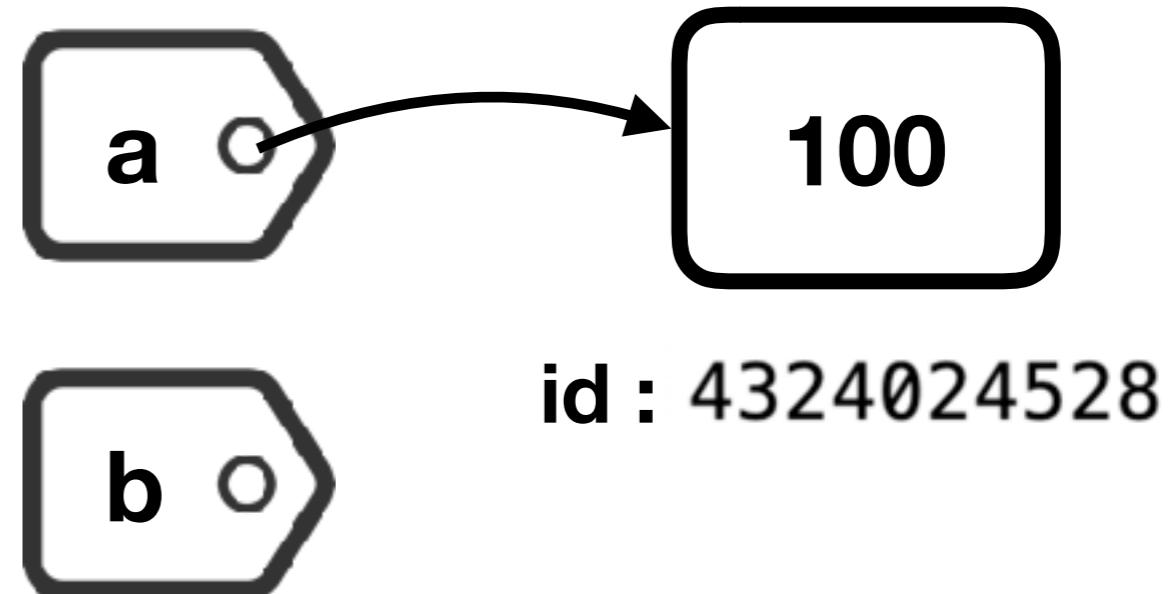
```
[>>> b = a
```

```
[>>> print(id(a))
```

```
4324024528
```

```
[>>> print(id(b))
```

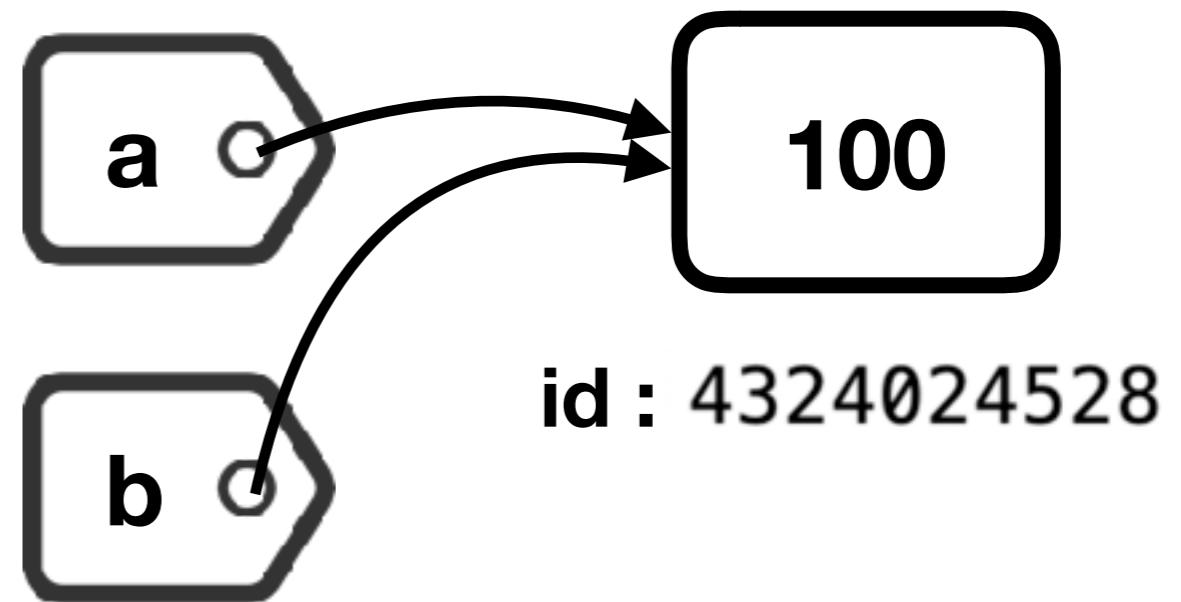
```
4324024528
```



객체를 여러 변수가 동시에 참조할 수 있다.

= (할당 연산자)는 객체에 대한 참조를 만들어 준다.

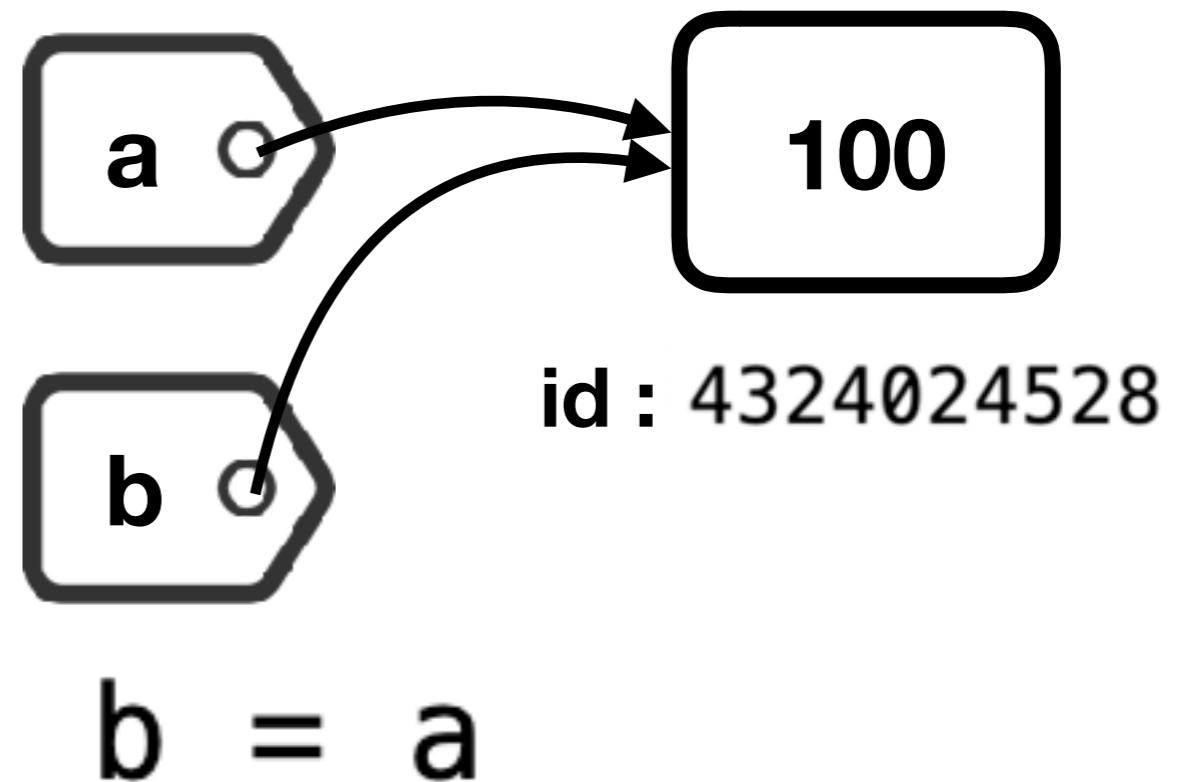
```
[>>> a = 100  
[>>> b = a  
[>>> print(id(a))  
4324024528  
[>>> print(id(b))  
4324024528
```



객체를 여러 변수가 동시에 참조할 수 있다.

= (할당 연산자)는 객체에 대한 참조를 만들어 준다.

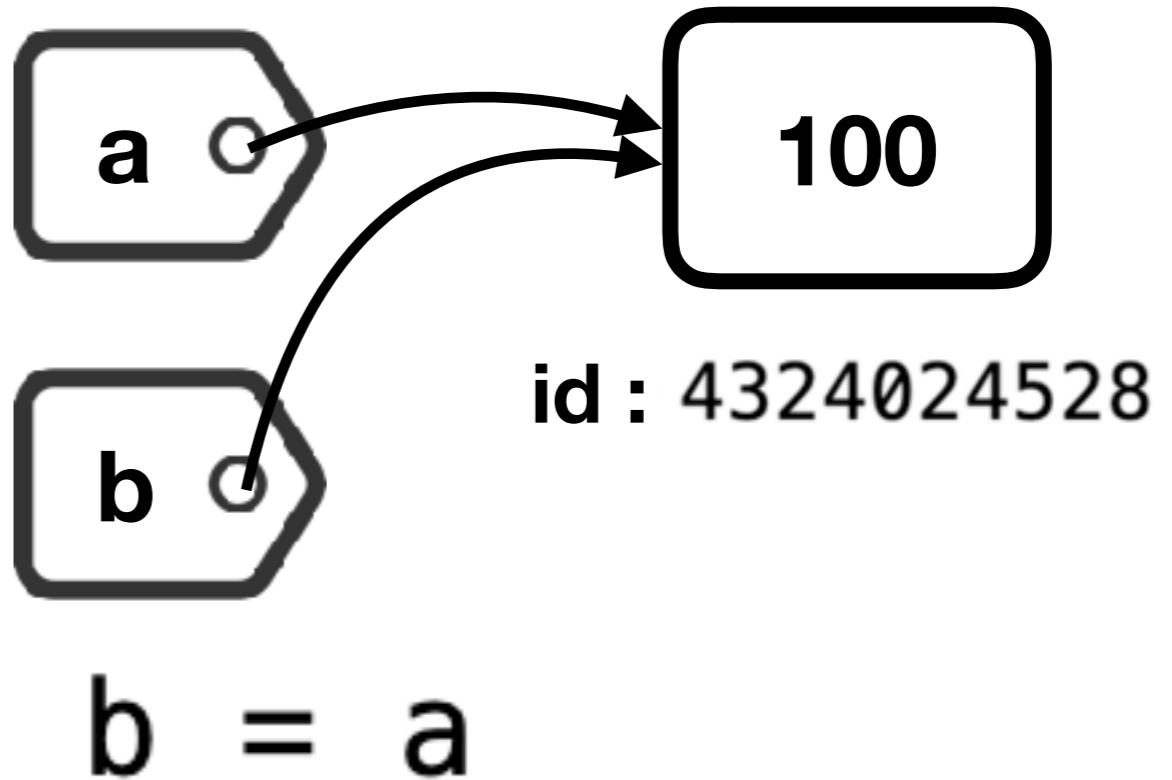
```
[>>> a = 100  
[>>> b = a  
[>>> print(id(a))  
4324024528  
[>>> print(id(b))  
4324024528
```



객체를 여러 변수가 동시에 참조할 수 있다.

= (할당 연산자)는 객체에 대한 참조를 만들어 준다.

```
[>>> a = 100  
[>>> b = a  
[>>> print(id(a))  
4324024528  
[>>> print(id(b))  
4324024528
```

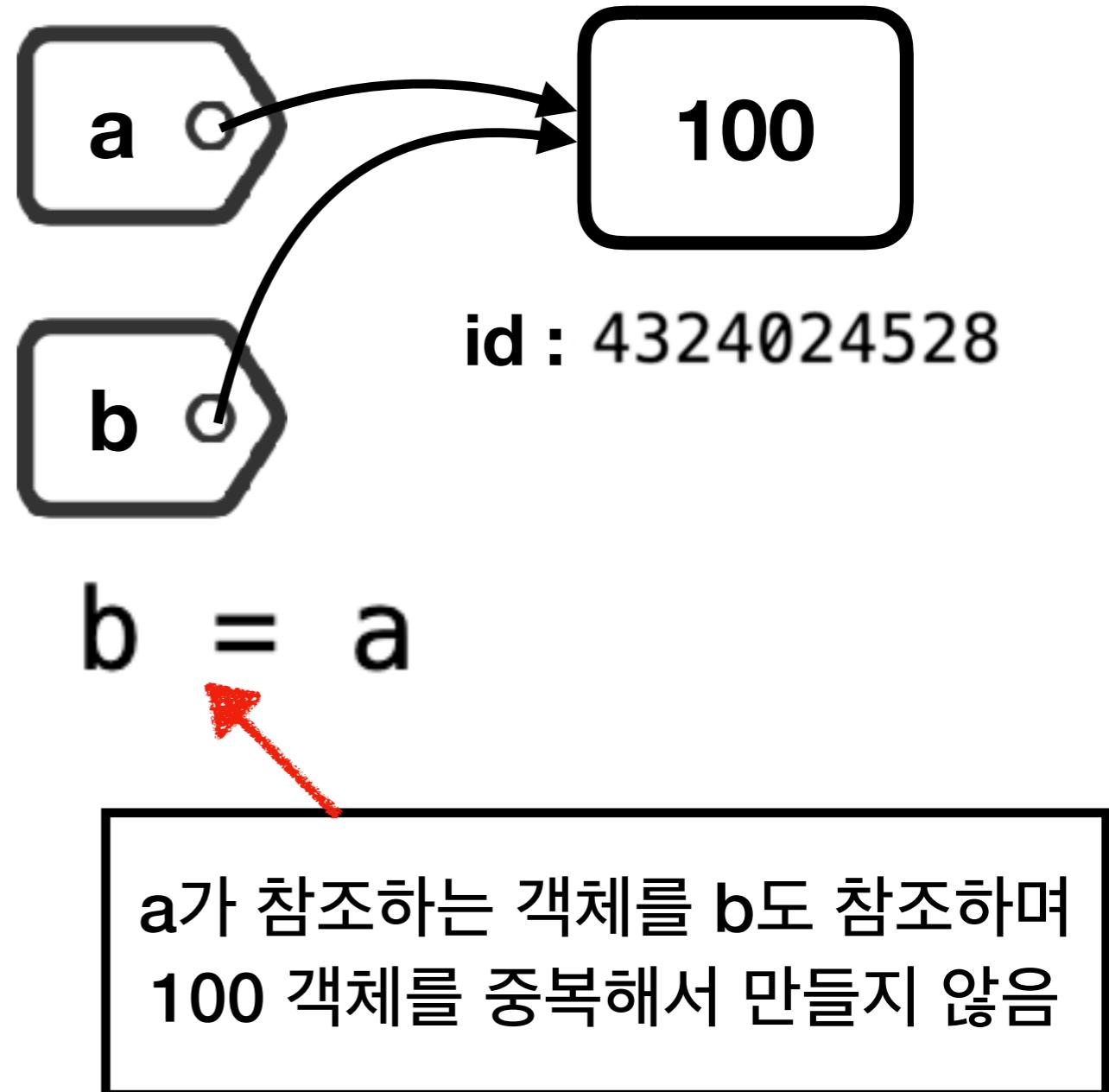


a가 참조하는 객체를 b도 참조하며
100 객체를 중복해서 만들지 않음

객체를 여러 변수가 동시에 참조할 수 있다.

= (할당 연산자)는 객체에 대한 참조를 만들어 준다.

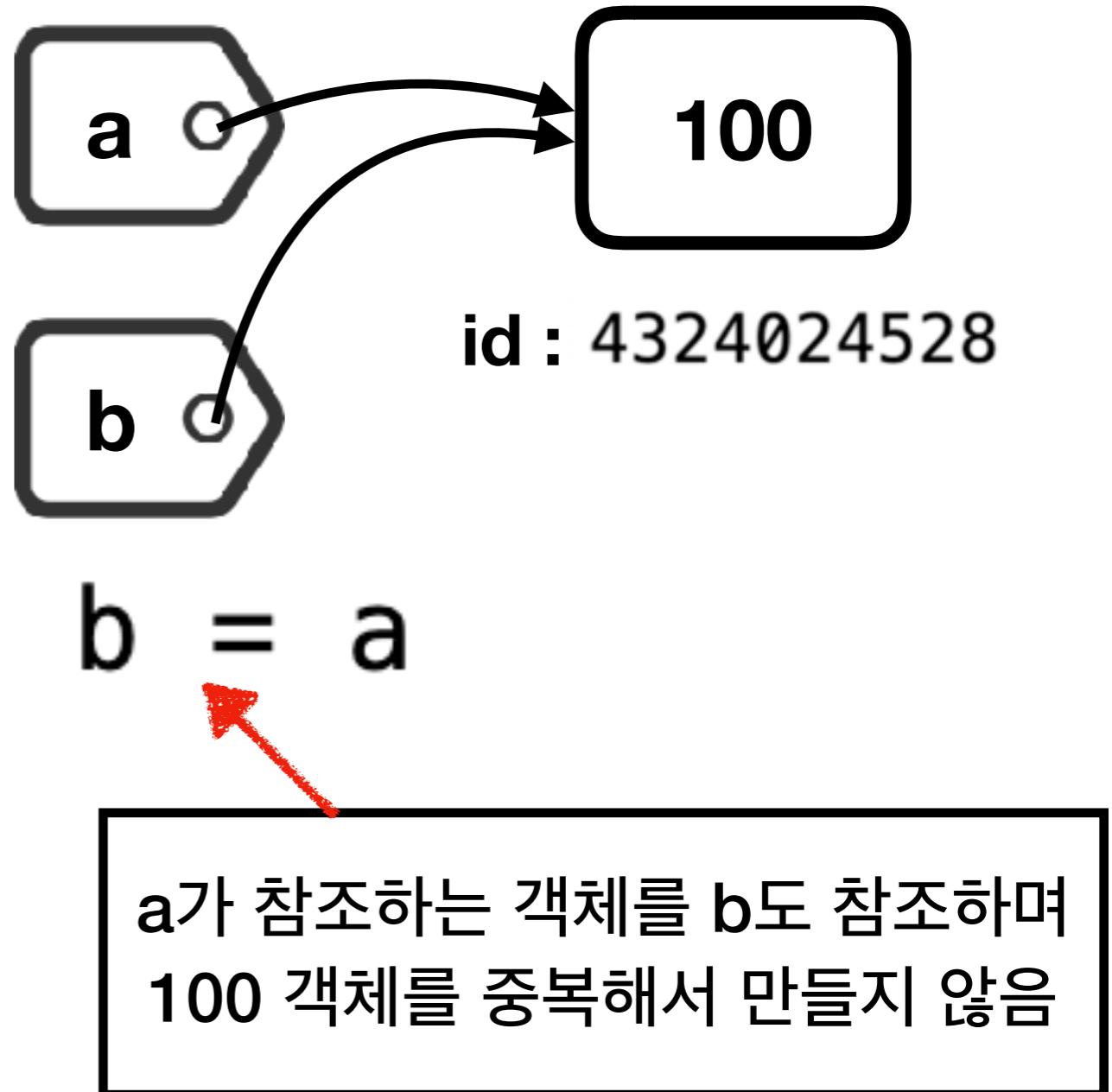
```
[>>> a = 100  
[>>> b = a  
[>>> print(id(a))  
4324024528  
[>>> print(id(b))  
4324024528
```



객체를 여러 변수가 동시에 참조할 수 있다.

= (할당 연산자)는 객체에 대한 참조를 만들어 준다.

```
[>>> a = 100  
[>>> b = a  
[>>> print(id(a))  
4324024528  
[>>> print(id(b))  
4324024528
```



C

변수 중심

Python

객체 중심

C

변수 중심

```
int a, b; // 정수형 변수 선언
```

Python

객체 중심

C

변수 중심

```
int a, b; // 정수형 변수 선언
```

a :



Python

객체 중심

C

변수 중심

```
int a, b; // 정수형 변수 선언
```

a :



b :



Python

객체 중심

C

변수 중심

```
int a, b; // 정수형 변수 선언
```

```
a = 100; // 정수값 할당
```

Python

객체 중심

a :



b :



C

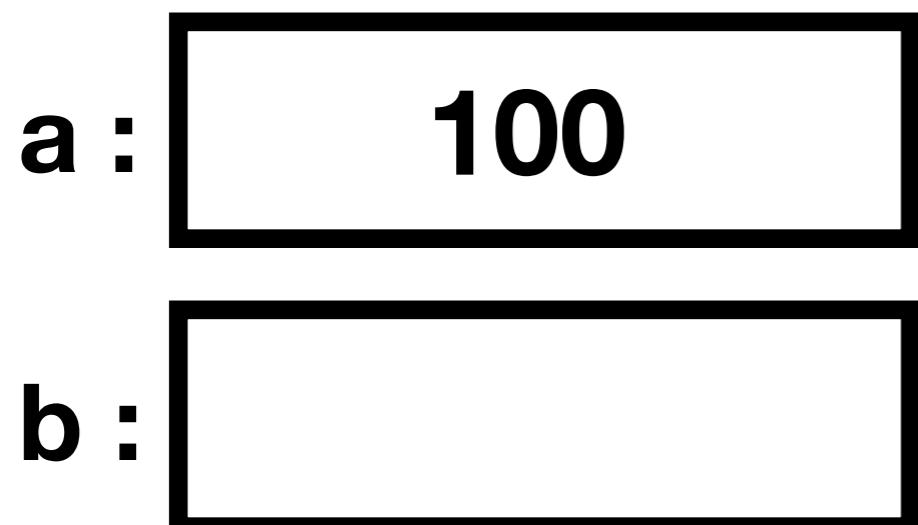
변수 중심

```
int a, b; // 정수형 변수 선언
```

```
a = 100; // 정수값 할당
```

Python

객체 중심



C

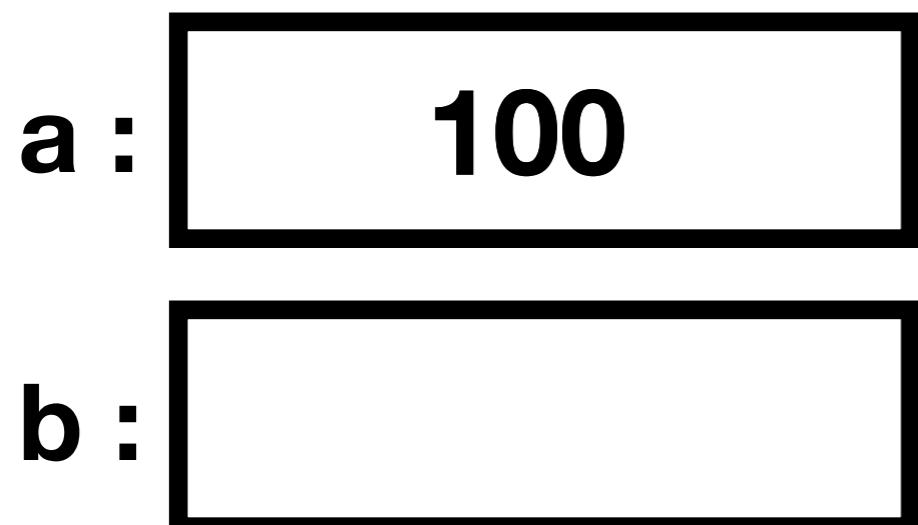
변수 중심

```
int a, b; // 정수형 변수 선언
```

```
a = 100; // 정수값 할당
```

Python

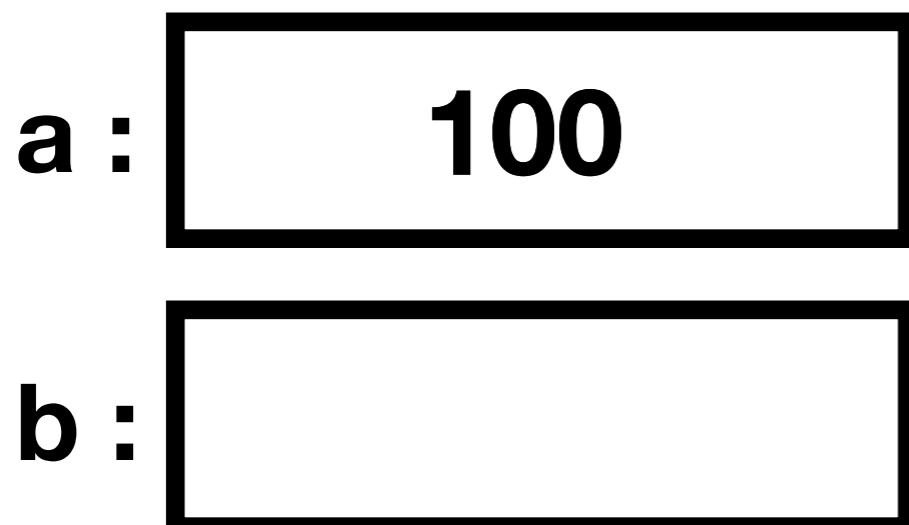
객체 중심



C

변수 중심

```
int a, b; // 정수형 변수 선언  
a = 100;    // 정수값 할당  
b = a;      // a의 값을 b에 할당
```



Python

객체 중심

```
int a, b; // 정수형 변수 선언  
a = 100;    // 정수값 할당  
b = a;      // a의 값을 b에 할당
```

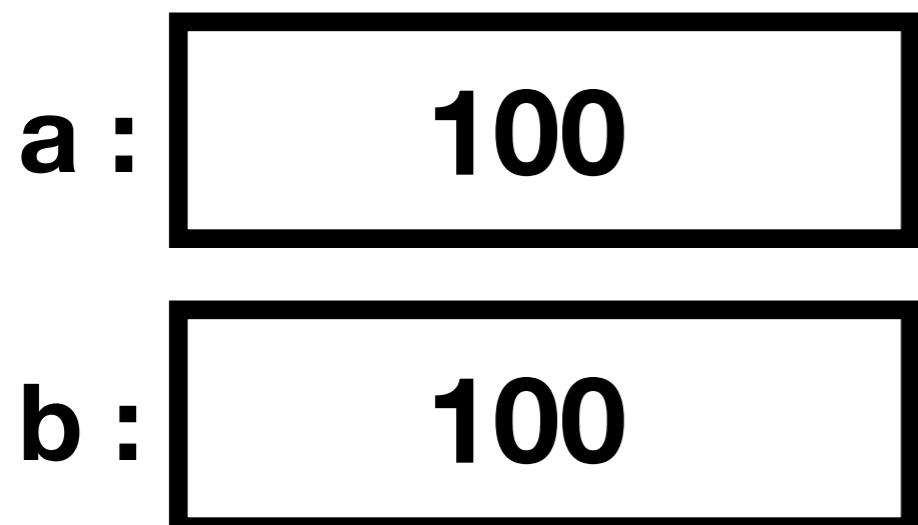
C

변수 중심

```
int a, b; // 정수형 변수 선언  
a = 100;    // 정수값 할당  
b = a;      // a의 값을 b에 할당
```

Python

객체 중심



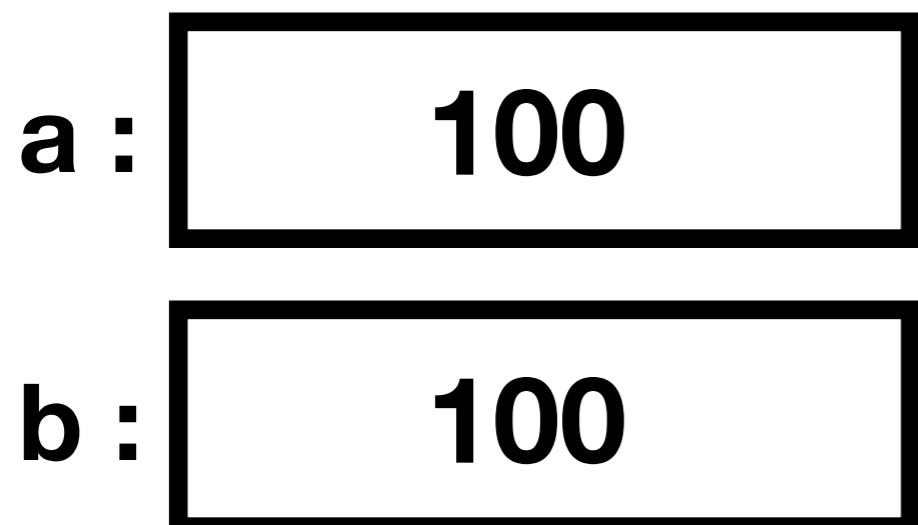
C

변수 중심

```
int a, b; // 정수형 변수 선언  
a = 100;    // 정수값 할당  
b = a;      // a의 값을 b에 할당
```

Python

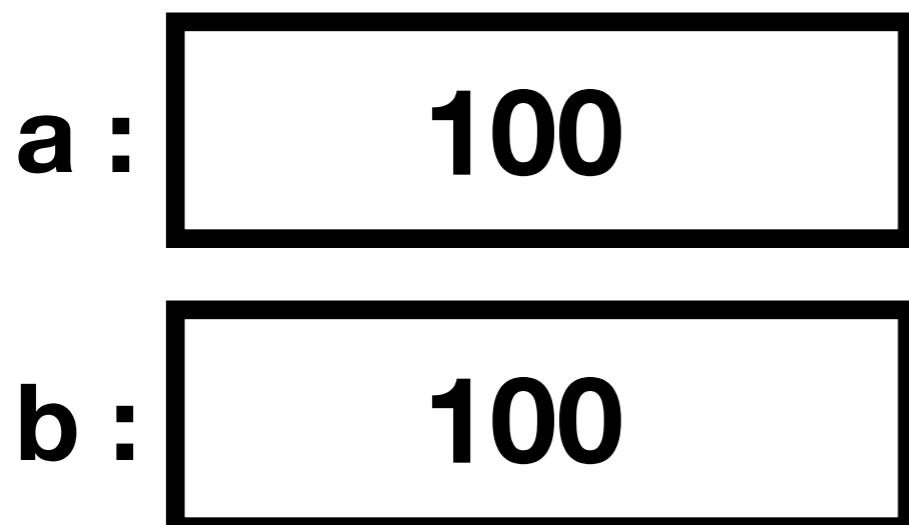
객체 중심



C

변수 중심

```
int a, b; // 정수형 변수 선언  
a = 100;    // 정수값 할당  
b = a;      // a의 값을 b에 할당
```



Python

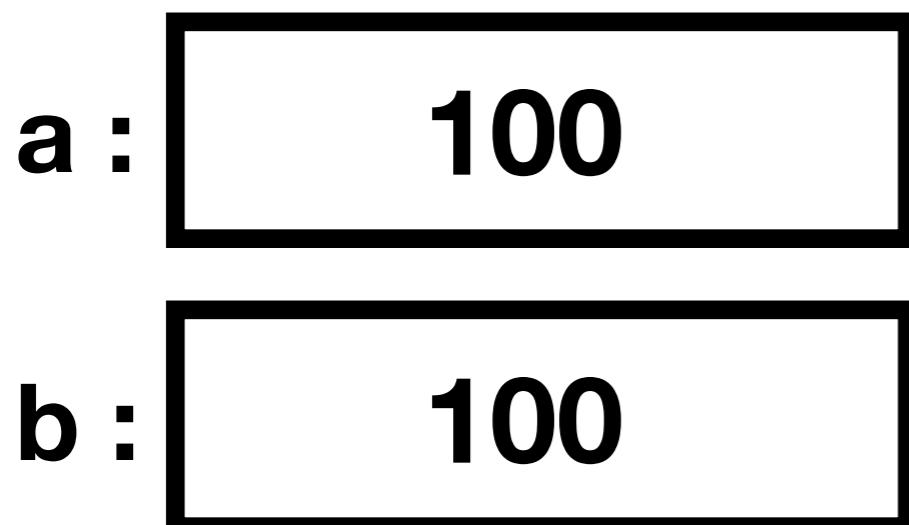
객체 중심

변수의 형 선언이 불필요함

C

변수 중심

```
int a, b; // 정수형 변수 선언  
a = 100; // 정수값 할당  
b = a; // a의 값을 b에 할당
```



Python

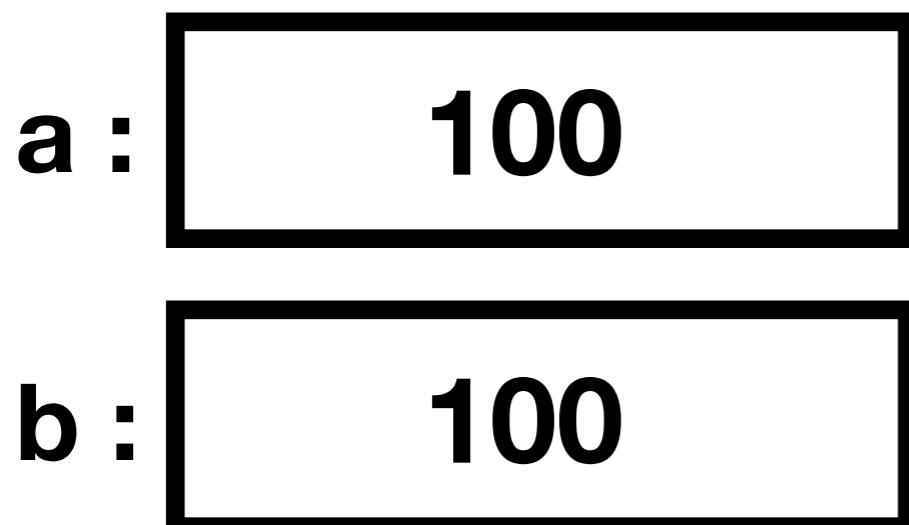
객체 중심

```
# 변수의 형 선언이 불필요함  
a = 100 # 100 객체 생성과 참조
```

C

변수 중심

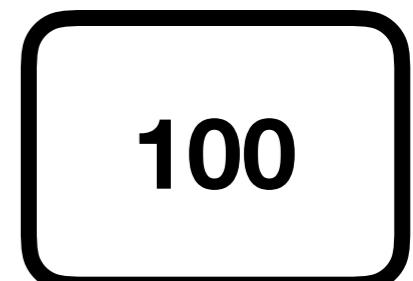
```
int a, b; // 정수형 변수 선언  
a = 100; // 정수값 할당  
b = a; // a의 값을 b에 할당
```



Python

객체 중심

```
# 변수의 형 선언이 불필요함  
a = 100 # 100 객체 생성과 참조
```

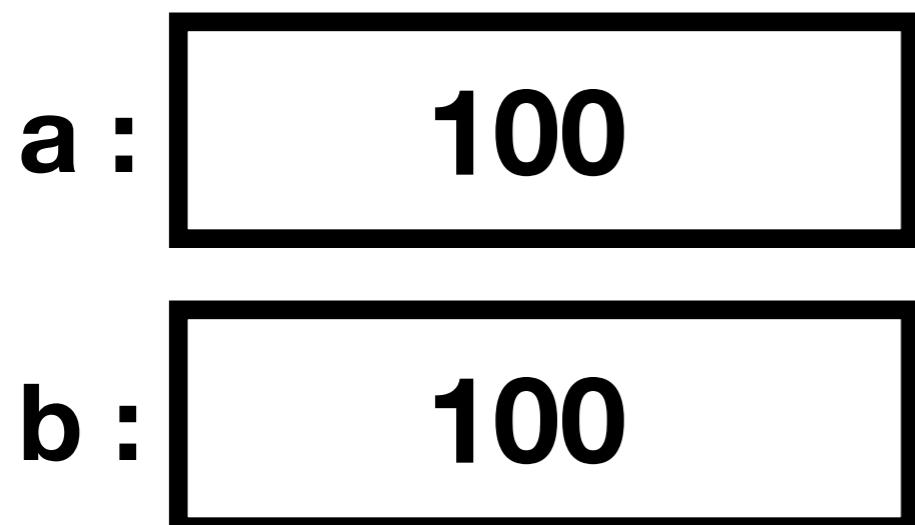


id : 4509016272

C

변수 중심

```
int a, b; // 정수형 변수 선언  
a = 100; // 정수값 할당  
b = a; // a의 값을 b에 할당
```



Python

객체 중심

```
# 변수의 형 선언이 불필요함  
a = 100 # 100 객체 생성과 참조
```

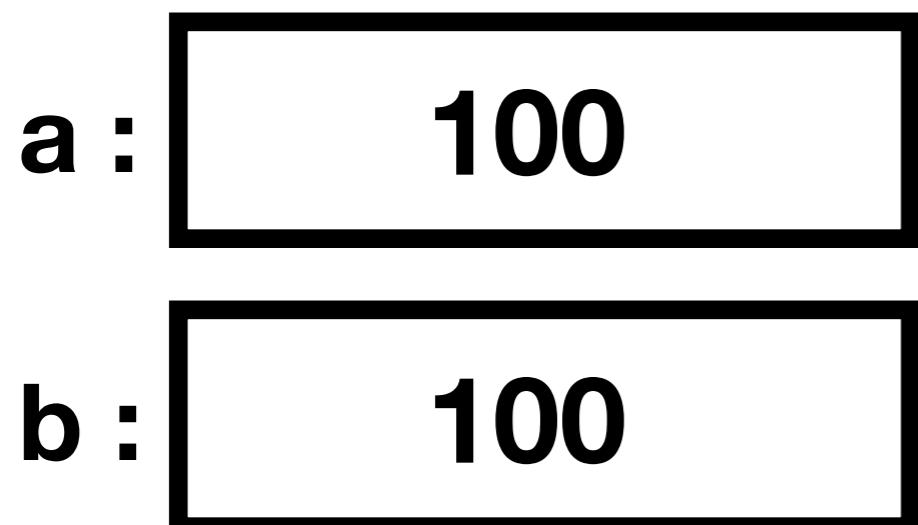


id : 4509016272

C

변수 중심

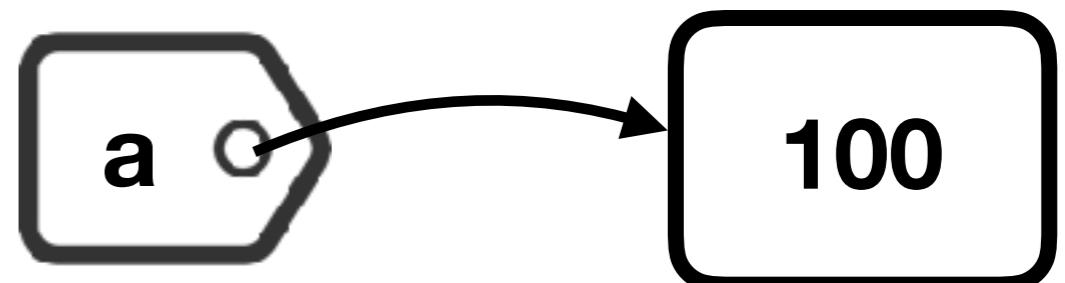
```
int a, b; // 정수형 변수 선언  
a = 100; // 정수값 할당  
b = a; // a의 값을 b에 할당
```



Python

객체 중심

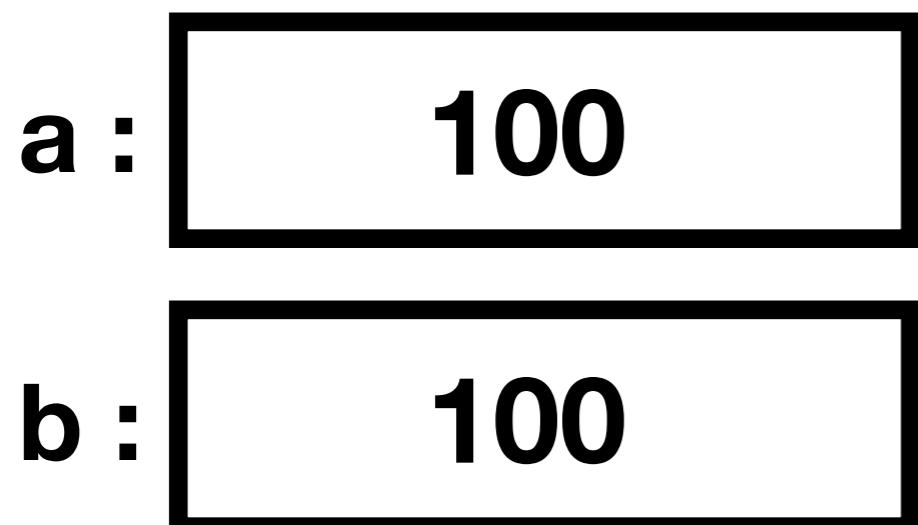
```
# 변수의 형 선언이 불필요함  
a = 100 # 100 객체 생성과 참조
```



C

변수 중심

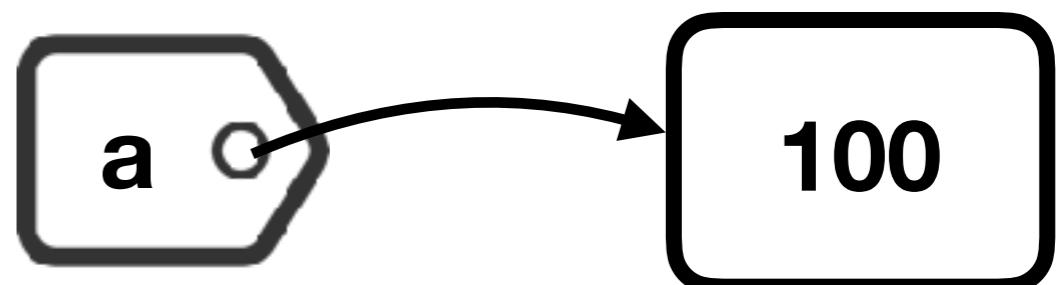
```
int a, b; // 정수형 변수 선언  
a = 100; // 정수값 할당  
b = a; // a의 값을 b에 할당
```



Python

객체 중심

```
# 변수의 형 선언이 불필요함  
a = 100 # 100 객체 생성과 참조  
b = a # 100 객체를 참조
```

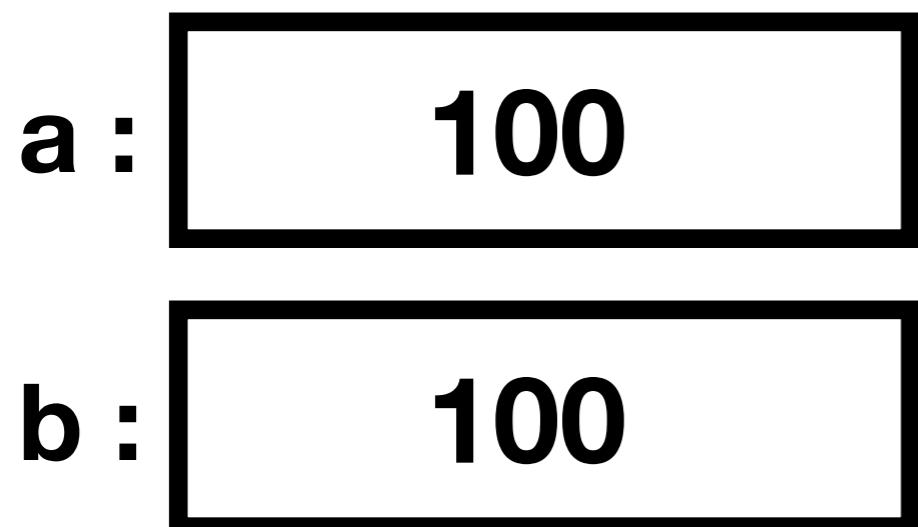


id : 4509016272

C

변수 중심

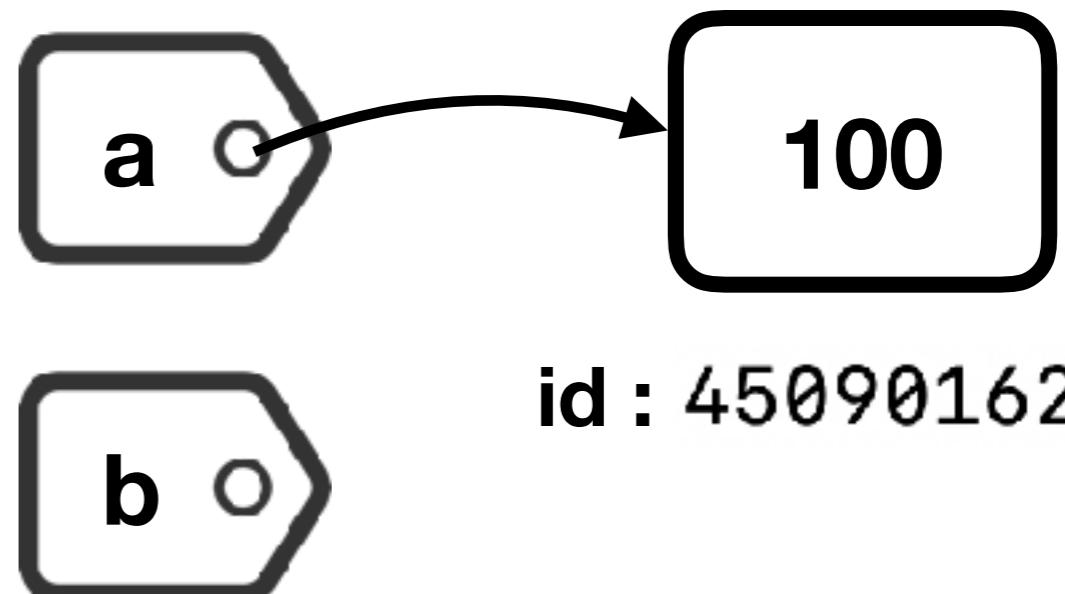
```
int a, b; // 정수형 변수 선언  
a = 100; // 정수값 할당  
b = a; // a의 값을 b에 할당
```



Python

객체 중심

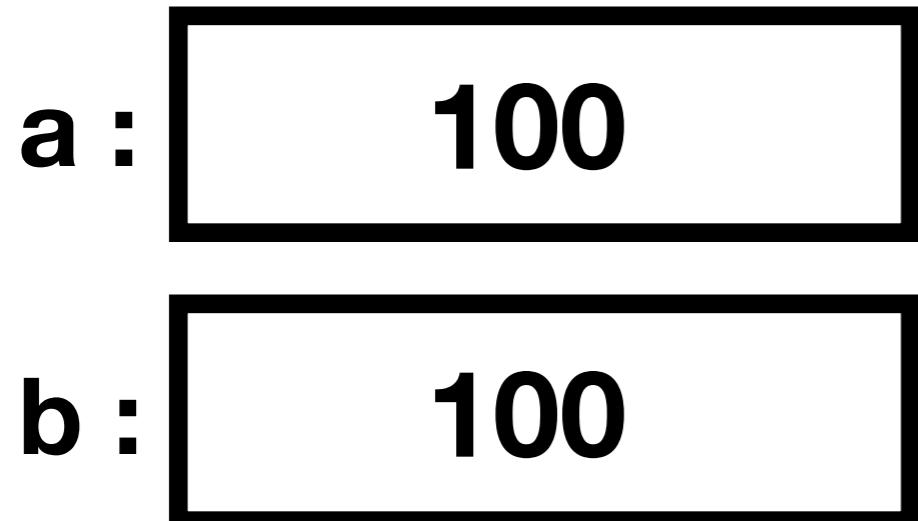
```
# 변수의 형 선언이 불필요함  
a = 100 # 100 객체 생성과 참조  
b = a # 100 객체를 참조
```



C

변수 중심

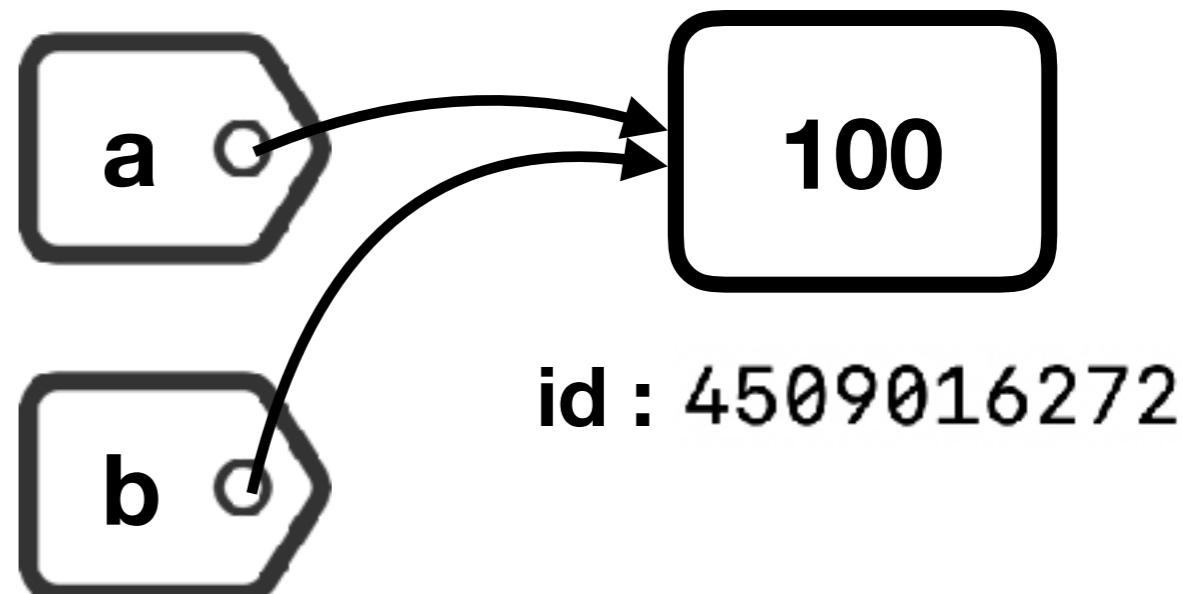
```
int a, b; // 정수형 변수 선언  
a = 100; // 정수값 할당  
b = a; // a의 값을 b에 할당
```



Python

객체 중심

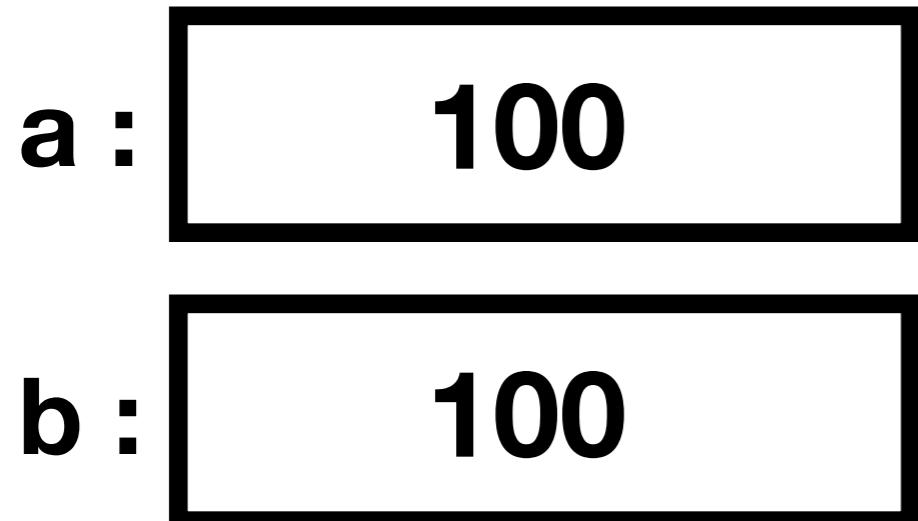
```
# 변수의 형 선언이 불필요함  
a = 100 # 100 객체 생성과 참조  
b = a # 100 객체를 참조
```



C

변수 중심

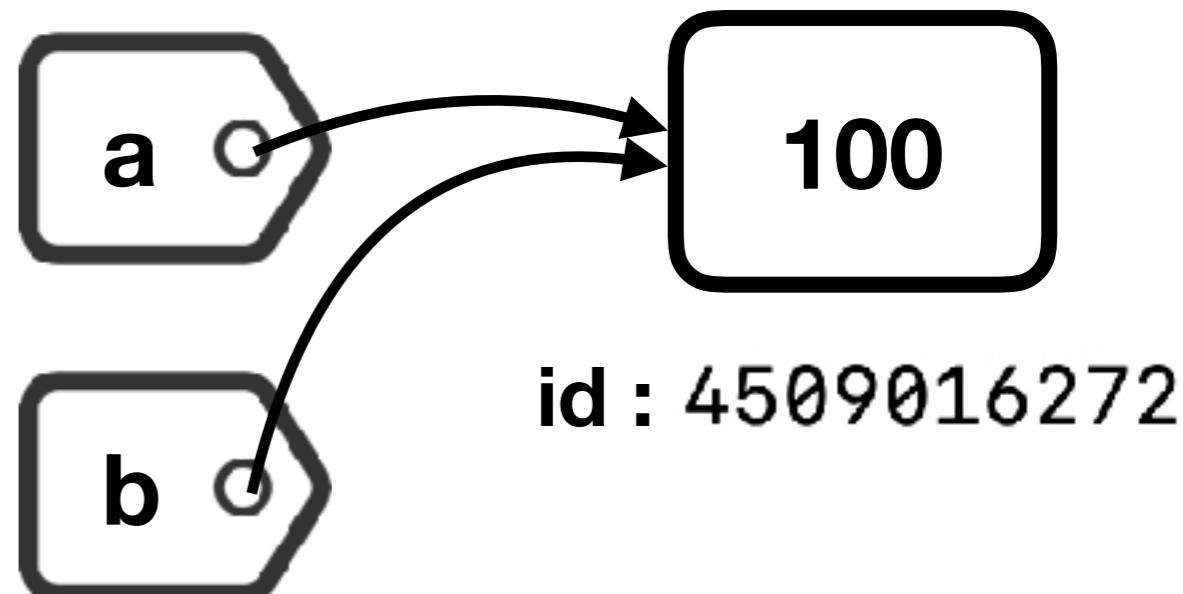
```
int a, b; // 정수형 변수 선언  
a = 100; // 정수값 할당  
b = a; // a의 값을 b에 할당
```



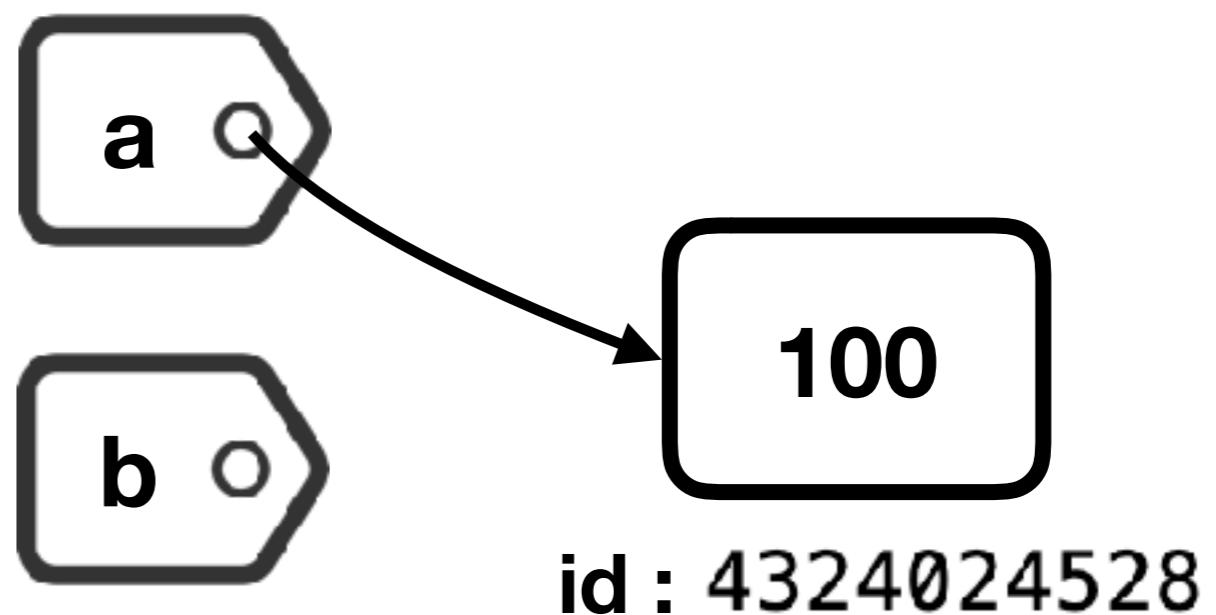
Python

객체 중심

```
# 변수의 형 선언이 불필요함  
a = 100 # 100 객체 생성과 참조  
b = a # 100 객체를 참조
```

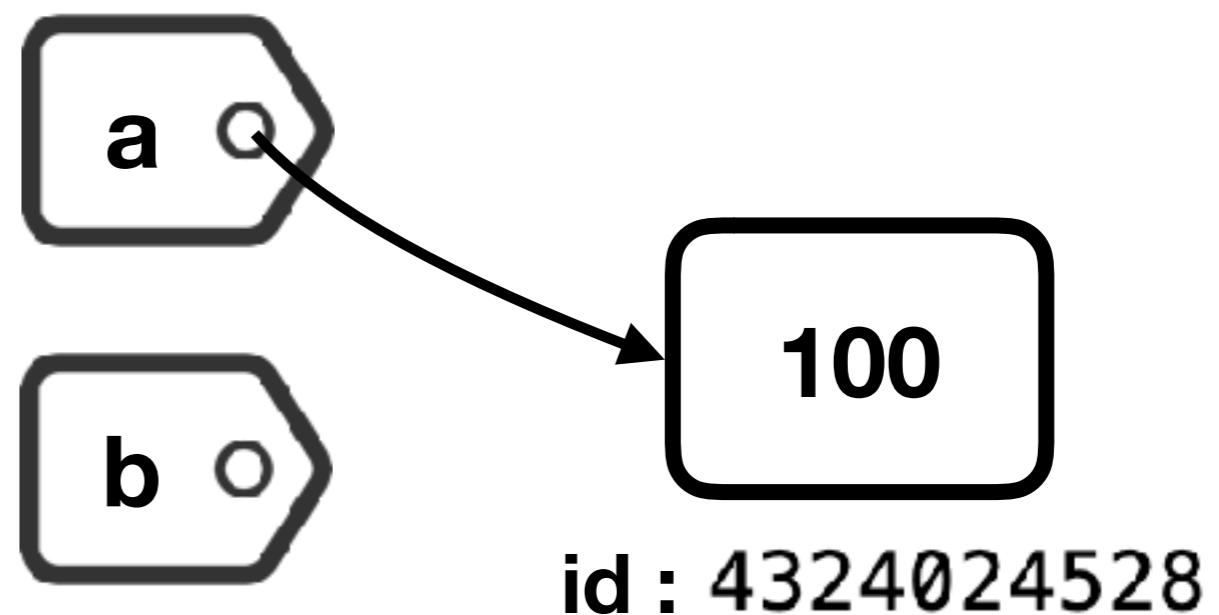


```
[>>> a = 100
[>>> b = a
[>>> a = 300
[>>> print(id(a))
4327017040
[>>> print(id(b))
4324024528
```



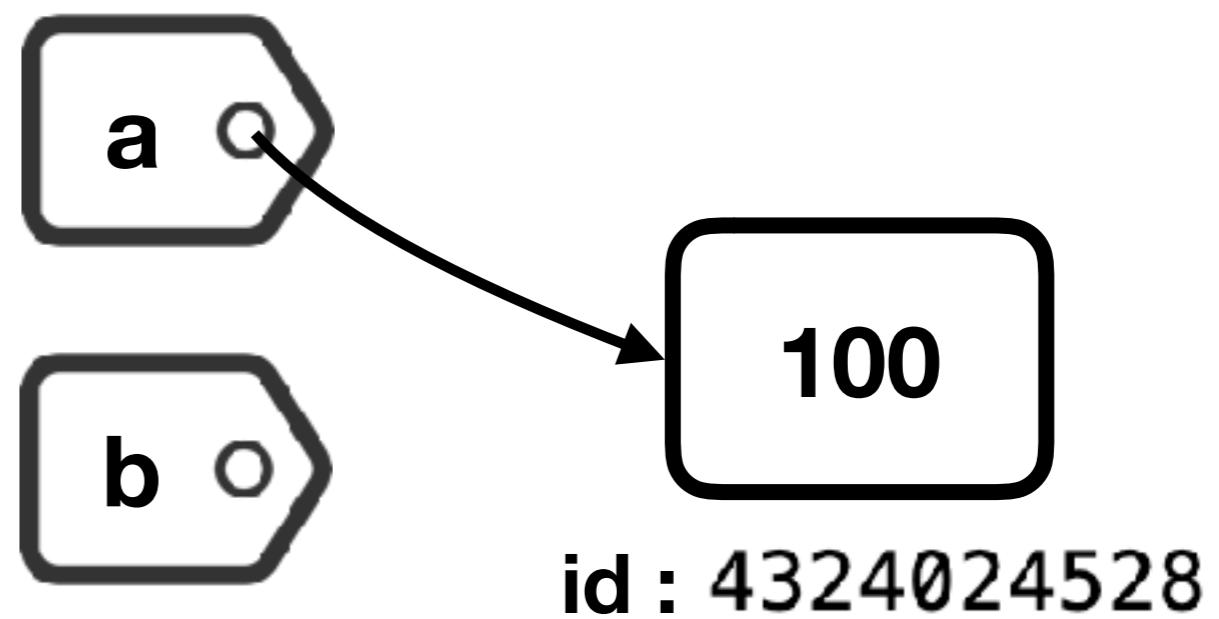
a는 새로운 객체를
참조할 수 있다

```
[>>> a = 100
[>>> b = a
[>>> a = 300
[>>> print(id(a))
4327017040
[>>> print(id(b))
4324024528
```



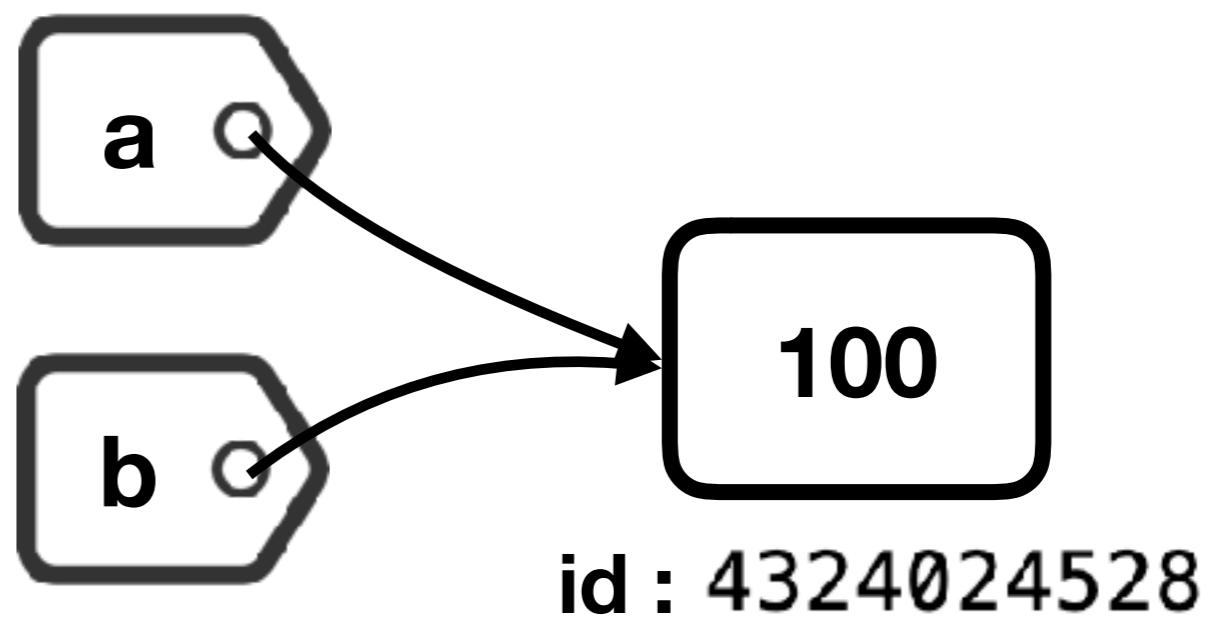
a는 새로운 객체를
참조할 수 있다

```
[>>> a = 100
[>>> b = a
[>>> a = 300
[>>> print(id(a))
4327017040
[>>> print(id(b))
4324024528
```



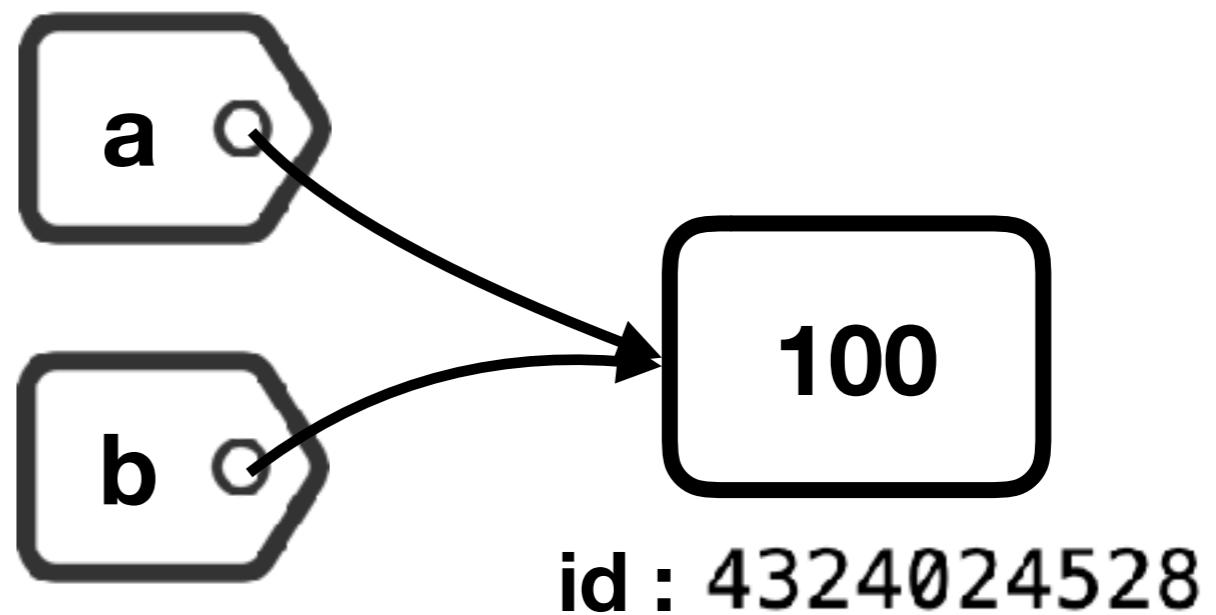
a는 새로운 객체를
참조할 수 있다

```
[>>> a = 100
[>>> b = a
[>>> a = 300
[>>> print(id(a))
4327017040
[>>> print(id(b))
4324024528
```



a는 새로운 객체를
참조할 수 있다

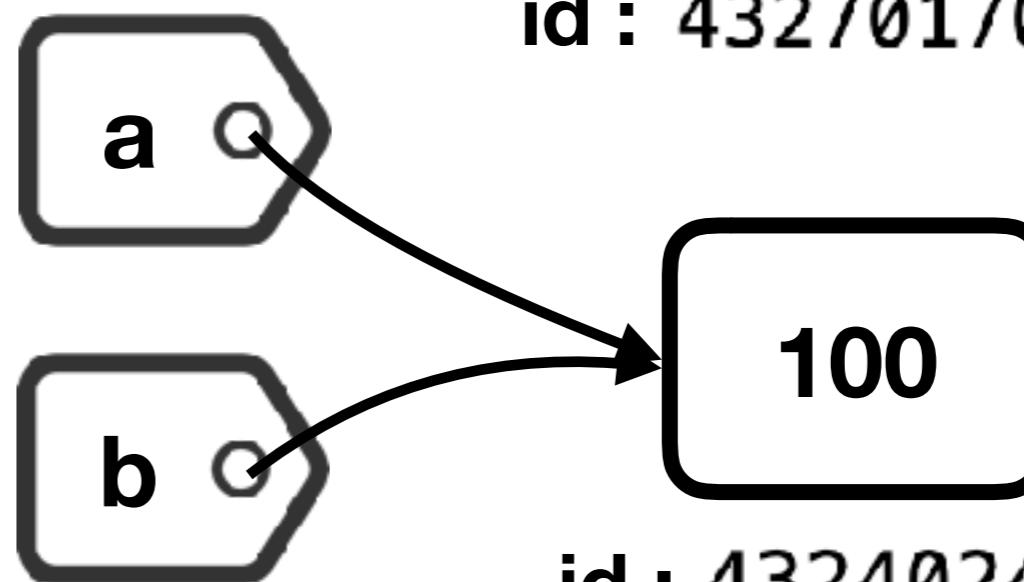
```
[>>> a = 100
[>>> b = a
[>>> a = 300
[>>> print(id(a))
4327017040
[>>> print(id(b))
4324024528
```



a는 새로운 객체를
참조할 수 있다

300

```
[>>> a = 100
[>>> b = a
[>>> a = 300
[>>> print(id(a))
4327017040
[>>> print(id(b))
4324024528
```



id : 4327017040

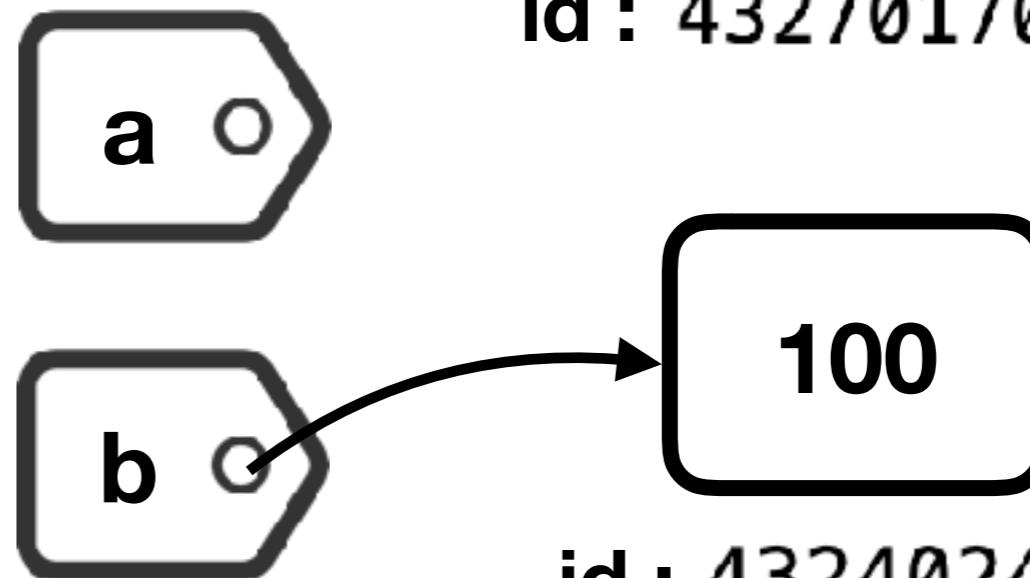
100

id : 4324024528

a는 새로운 객체를
참조할 수 있다

300

```
[>>> a = 100
[>>> b = a
[>>> a = 300
[>>> print(id(a))
4327017040
[>>> print(id(b))
4324024528
```



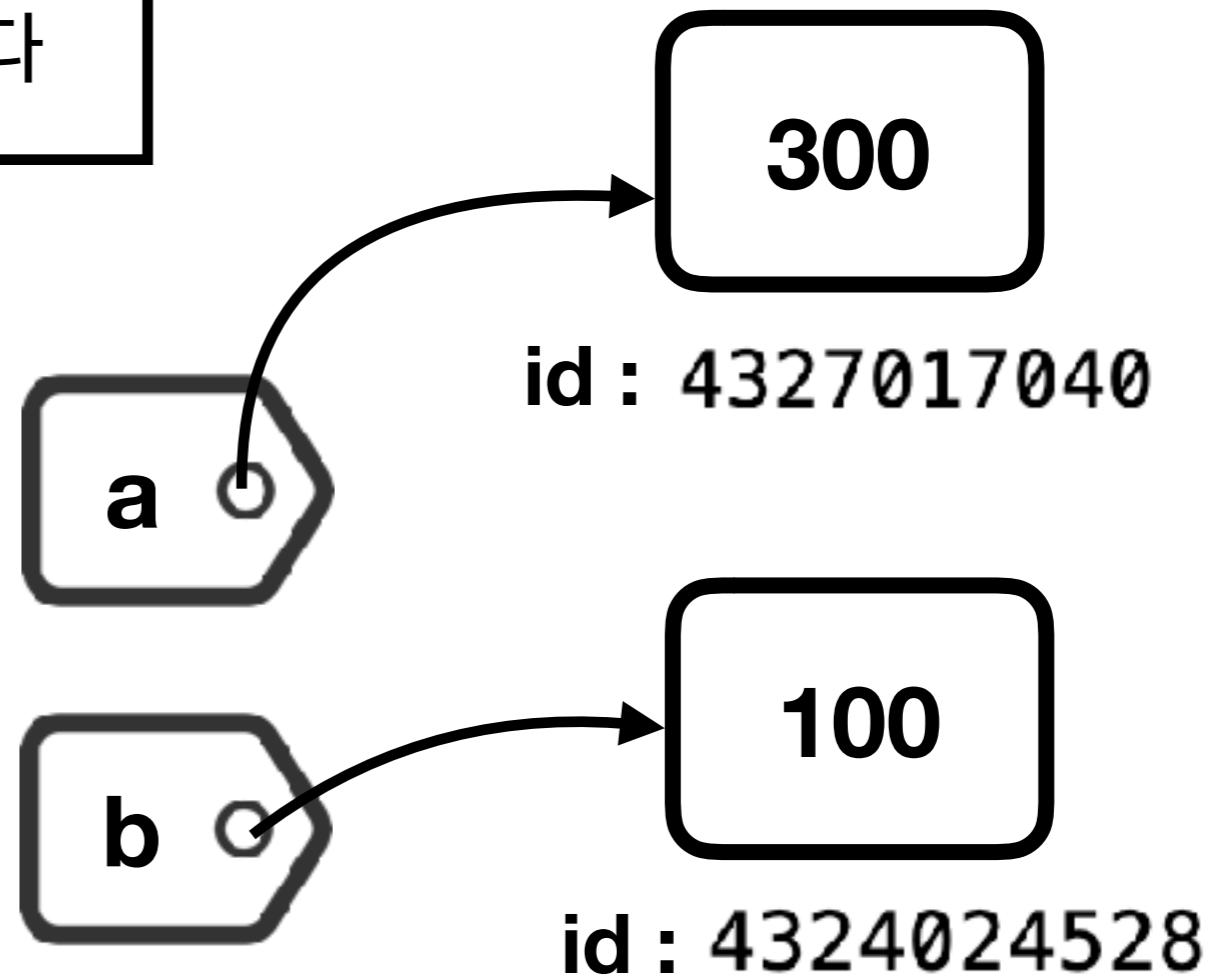
id : 4327017040

100

id : 4324024528

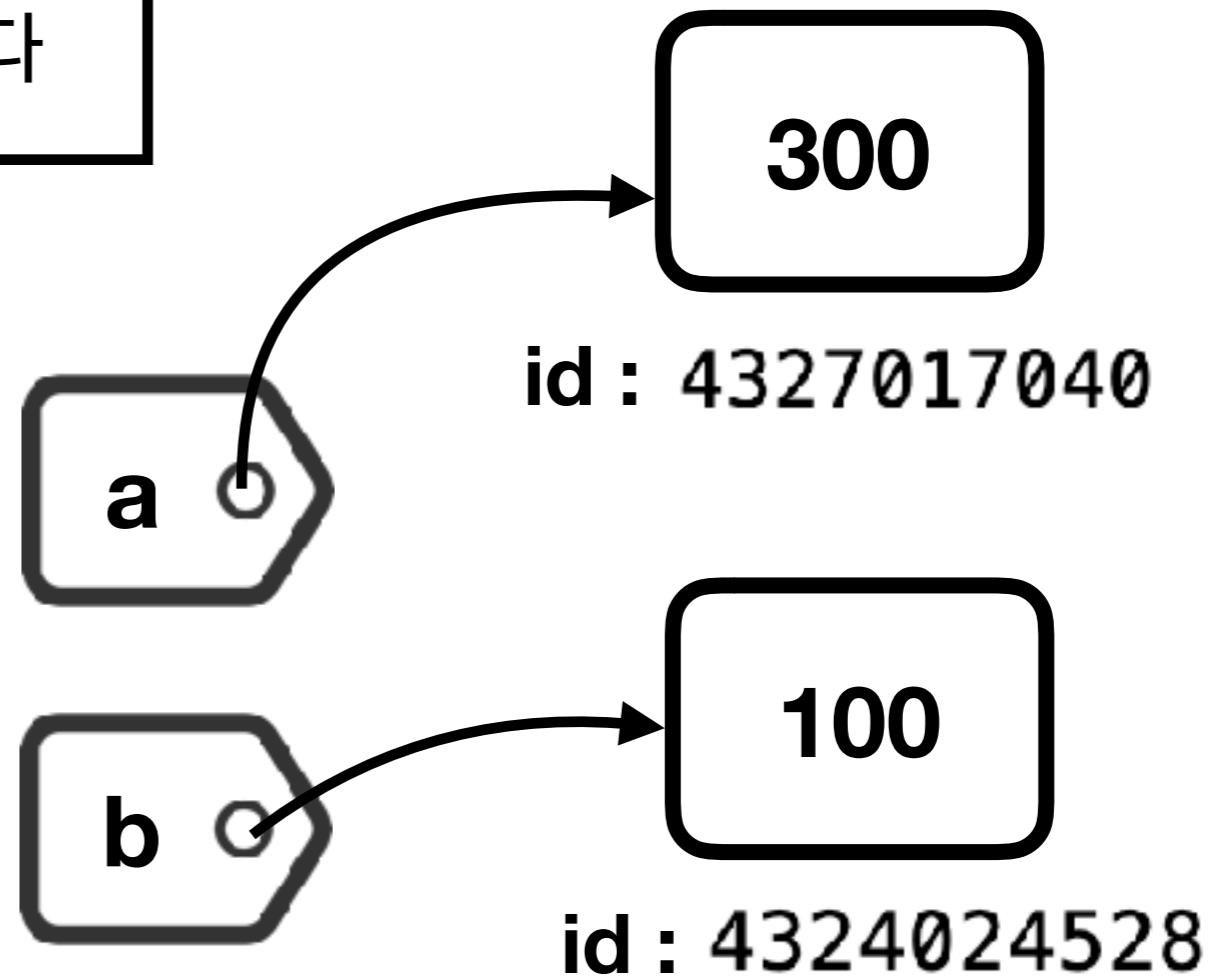
a는 새로운 객체를
참조할 수 있다

```
[>>> a = 100
[>>> b = a
[>>> a = 300
[>>> print(id(a))
4327017040
[>>> print(id(b))
4324024528
```



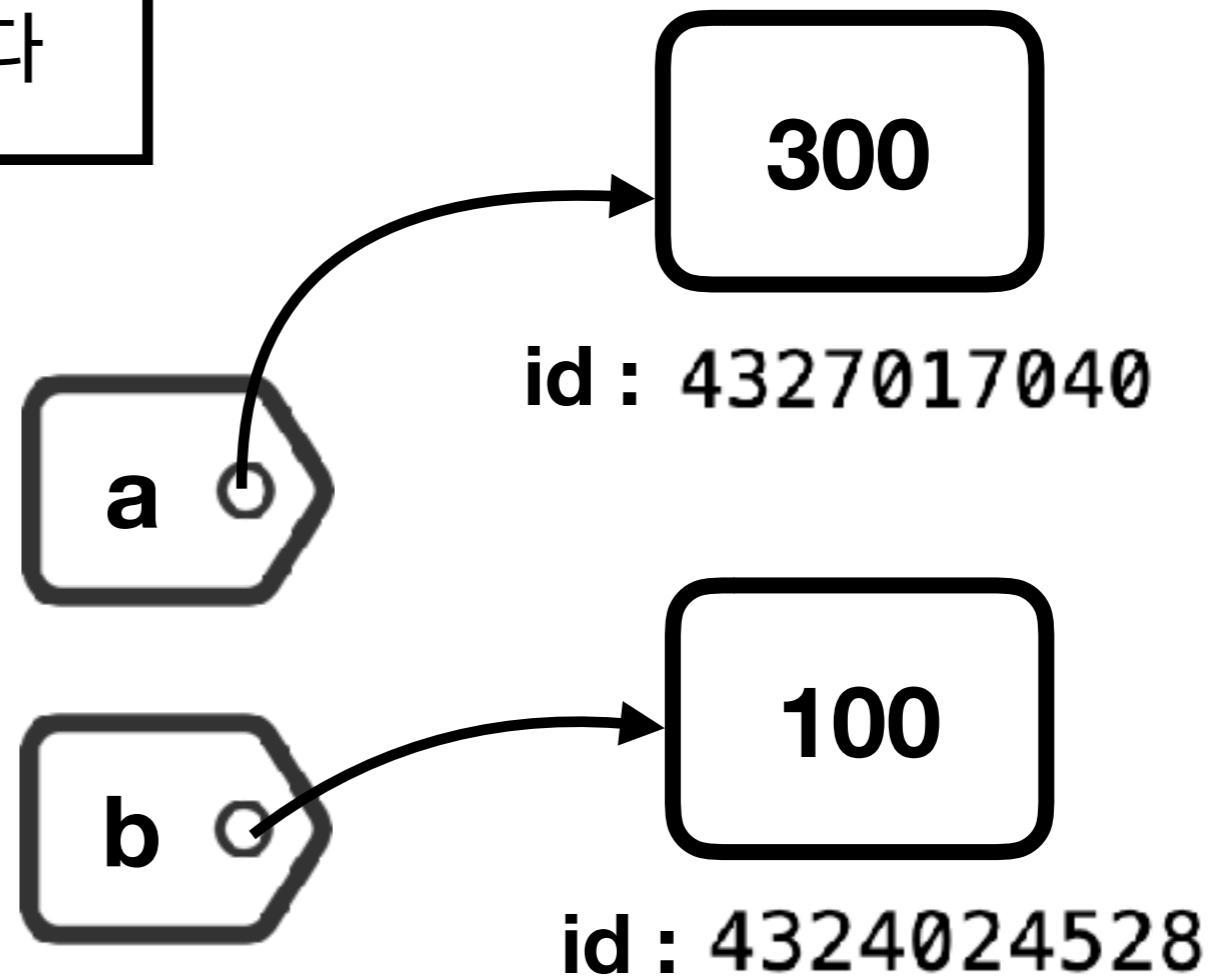
a는 새로운 객체를
참조할 수 있다

```
[>>> a = 100
[>>> b = a
[>>> a = 300
[>>> print(id(a))
4327017040
[>>> print(id(b))
4324024528
```



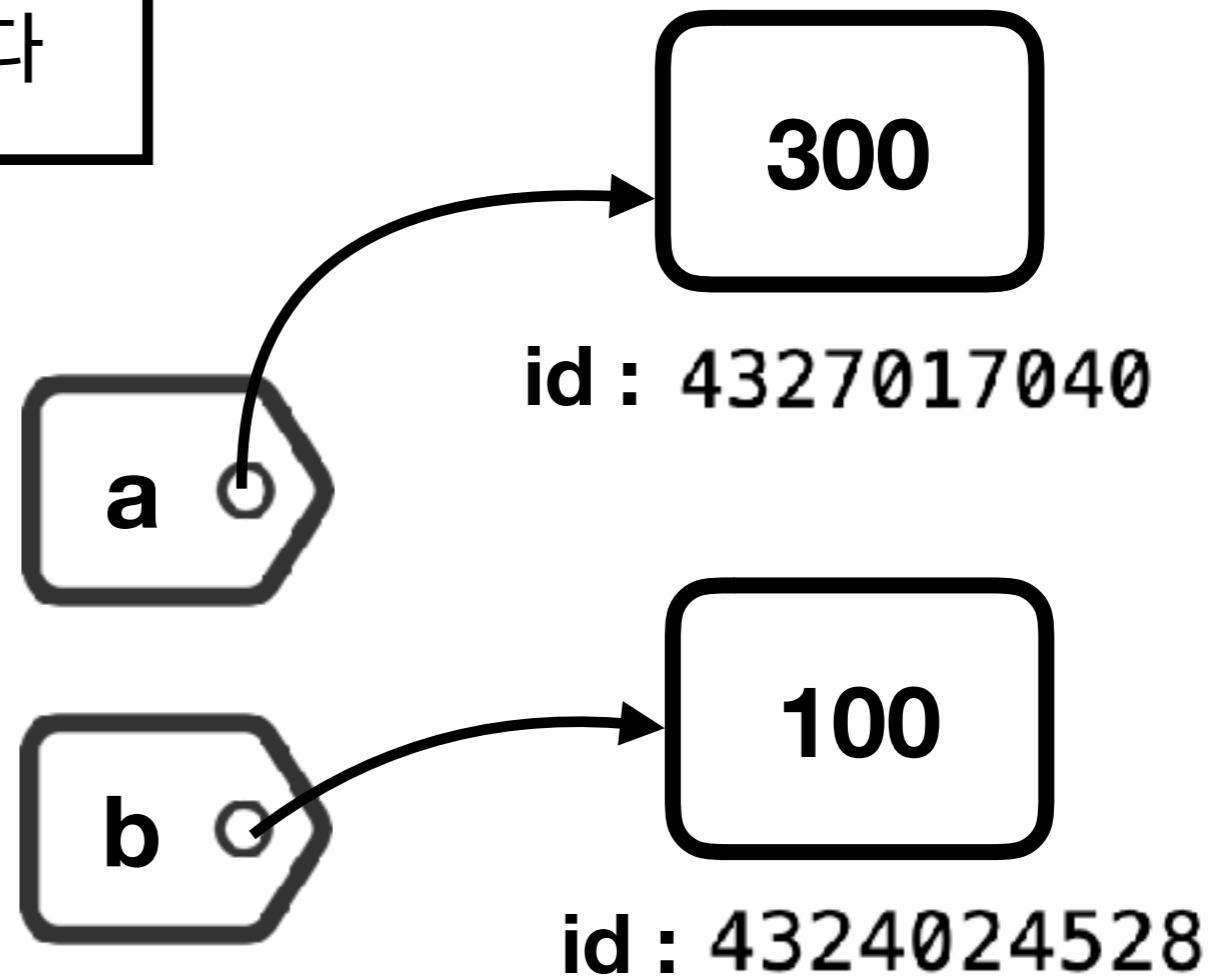
a는 새로운 객체를
참조할 수 있다

```
[>>> a = 100
[>>> b = a
[>>> a = 300
[>>> print(id(a))
4327017040
[>>> print(id(b))
4324024528
```



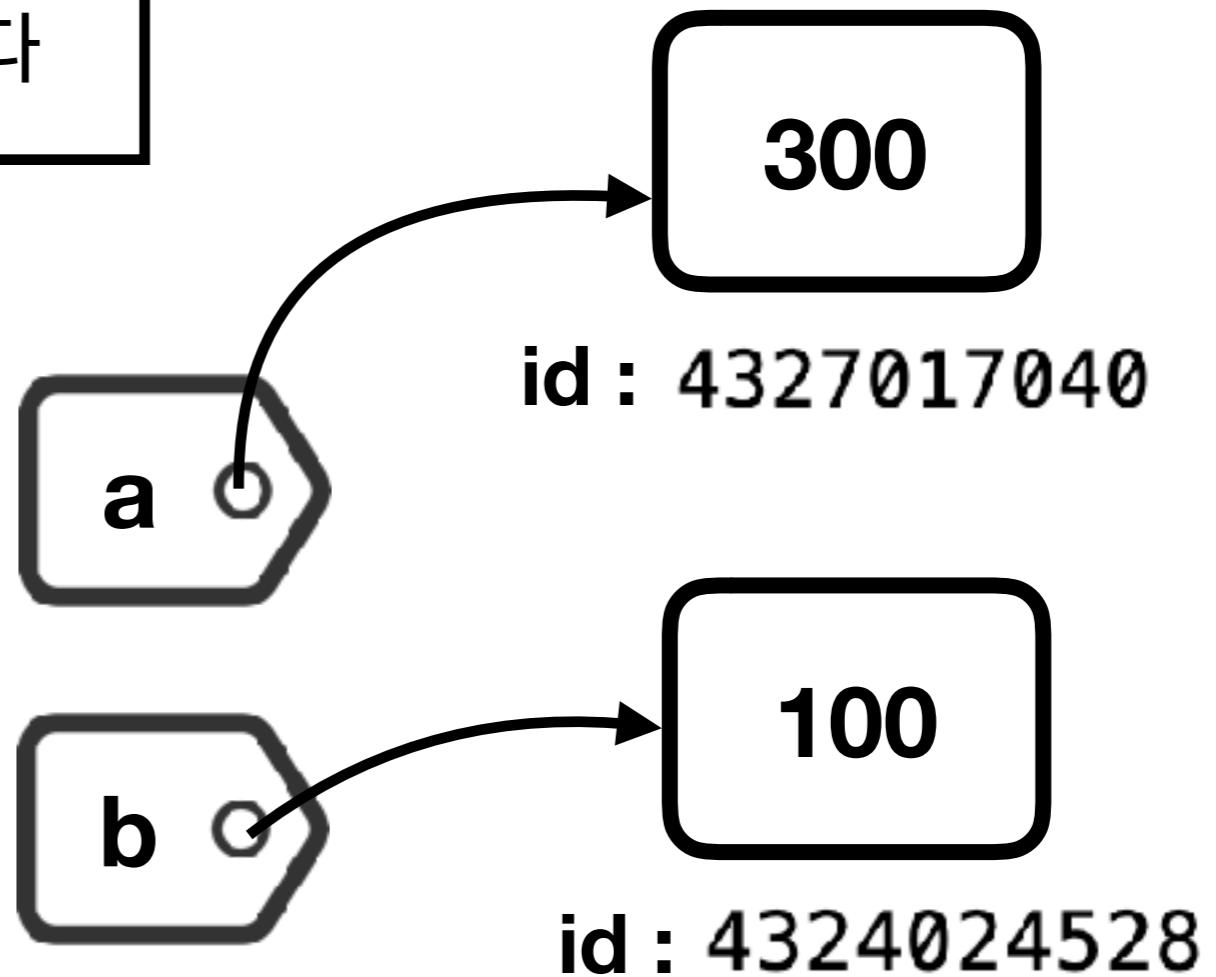
a는 새로운 객체를
참조할 수 있다

```
[>>> a = 100
[>>> b = a
[>>> a = 300
[>>> print(id(a))
4327017040
[>>> print(id(b))
4324024528
```



a는 새로운 객체를
참조할 수 있다

```
[>>> a = 100
[>>> b = a
[>>> a = 300
[>>> print(id(a))
4327017040
[>>> print(id(b))
4324024528
```



Lab

리스트 자료형

리스트

```
>>> list1 = ["one", "two", 3, 4]
>>> list2 = [1, 2, 3, 4]
>>> list3 = ["one", "two", "three", "four"]
>>> list3[1]
'two'
```

리스트

- 파이썬의 리스트는 타 언어의 배열과 비슷해 보인다(??).

```
>>> list1 = ["one", "two", 3, 4]
>>> list2 = [1, 2, 3, 4]
>>> list3 = ["one", "two", "three", "four"]
>>> list3[1]
'two'
```

리스트

- 파이썬의 리스트는 타 언어의 배열과 비슷해 보인다(??).
- 하지만 하나의 리스트에 서로 다른 자료형의 항목을 포함할 수 있다.(리스트, 딕셔너리 등을 포함할 수 있다)

```
>>> list1 = ["one", "two", 3, 4]
>>> list2 = [1, 2, 3, 4]
>>> list3 = ["one", "two", "three", "four"]
>>> list3[1]
'two'
```

리스트

- 파이썬의 리스트는 타 언어의 배열과 비슷해 보인다(??).
- 하지만 하나의 리스트에 서로 다른 자료형의 항목을 포함할 수 있다.(리스트, 딕셔너리 등을 포함할 수 있다)
- 매우 강력한 기능이 있는 자료형이다.

```
>>> list1 = ["one", "two", 3, 4]
>>> list2 = [1, 2, 3, 4]
>>> list3 = ["one", "two", "three", "four"]
>>> list3[1]
'two'
```

리스트

리스트

- 배열과 같은 인덱싱 가능한데다 슬라이싱도 지원하고

리스트

- 배열과 같은 인덱싱 가능한데다 슬라이싱도 지원하고
- sort() 도 가능하다

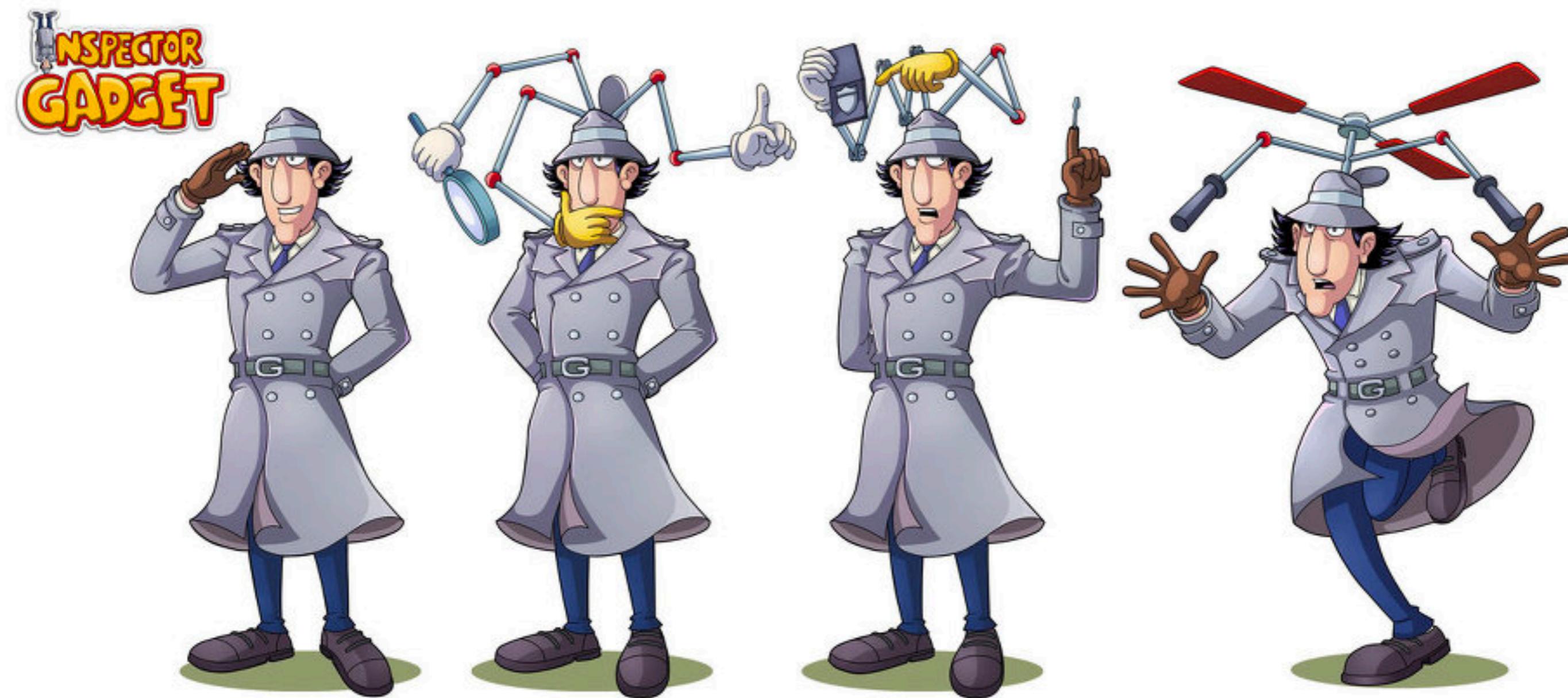
리스트

- 배열과 같은 인덱싱 가능한데다 슬라이싱도 지원하고
- `sort()` 도 가능하다
- 추가, 삭제 뿐만 아니라 스택 자료구조에 있는 `pop()` 도 지원하고

리스트

- 배열과 같은 인덱싱 가능한데다 슬라이싱도 지원하고
- sort() 도 가능하다
- 추가, 삭제 뿐만 아니라 스택 자료구조에 있는 pop() 도 지원하고
- count()를 이용하여 리스트에 있는 중복 원소의 갯수도 알려준다

리스트



INSPECTOR GADGET
CHARACTER LINEUP
ART: José Cobán

리스트



리스트가 만능인 이유?

GADGET
JP

C

배열

Python

리스트

C

배열

// 배열의 선언과 초기화

```
int a[3] = {100, 200, 300};
```

Python

리스트

C

배열

// 배열의 선언과 초기화

```
int a[3] = {100, 200, 300};
```

a



Python

리스트

C

배열

// 배열의 선언과 초기화

```
int a[3] = {100, 200, 300};
```

a

100	200	300
-----	-----	-----

Python

리스트

C

배열

// 배열의 선언과 초기화

```
int a[3] = {100, 200, 300};
```

a

100	200	300
-----	-----	-----

배열 원소의 크기는 int 형으로 고정됨

Python

리스트

C

배열

// 배열의 선언과 초기화

```
int a[3] = {100, 200, 300};
```

a

100	200	300
-----	-----	-----

배열 원소의 크기는 int 형으로 고정됨

Python

리스트

C

배열

// 배열의 선언과 초기화

```
int a[3] = {100, 200, 300};
```

a

100	200	300
-----	-----	-----

배열 원소의 크기는 int 형으로 고정됨

Python

리스트

리스트 객체 생성

```
a = [100, 200, 300]
```

C

배열

// 배열의 선언과 초기화

```
int a[3] = {100, 200, 300};
```

a



배열 원소의 크기는 int 형으로 고정됨

Python

리스트

리스트 객체 생성

```
a = [100, 200, 300]
```



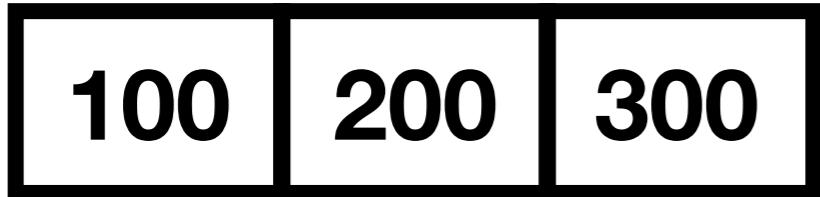
C

배열

// 배열의 선언과 초기화

```
int a[3] = {100, 200, 300};
```

a



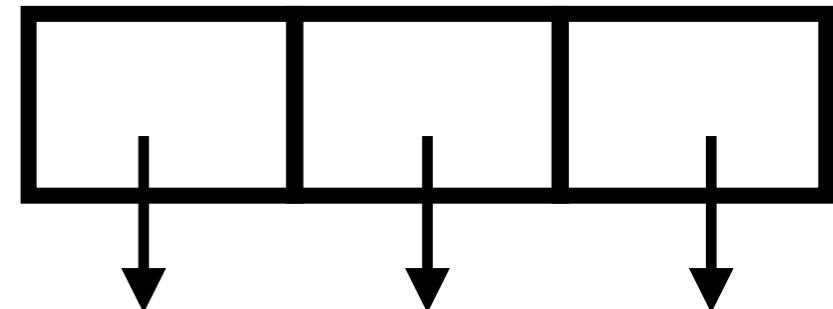
배열 원소의 크기는 int 형으로 고정됨

Python

리스트

리스트 객체 생성

```
a = [100, 200, 300]
```



C

배열

// 배열의 선언과 초기화

```
int a[3] = {100, 200, 300};
```

a



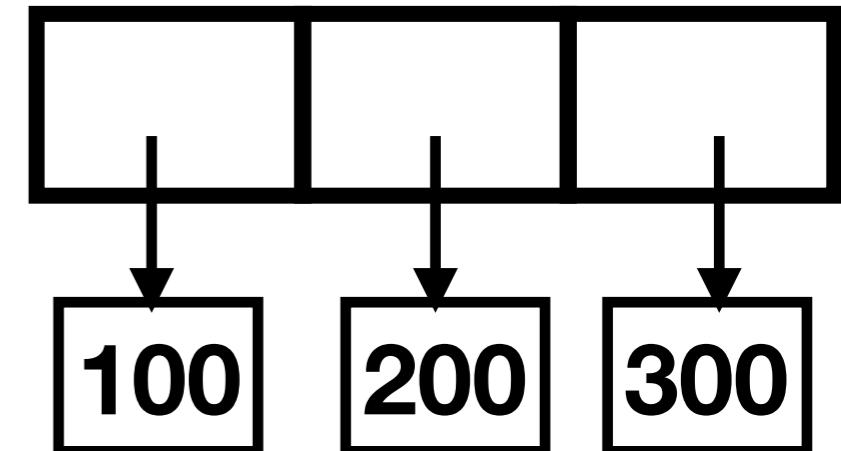
배열 원소의 크기는 int 형으로 고정됨

Python

리스트

리스트 객체 생성

```
a = [100, 200, 300]
```



C

배열

// 배열의 선언과 초기화

```
int a[3] = {100, 200, 300};
```

a



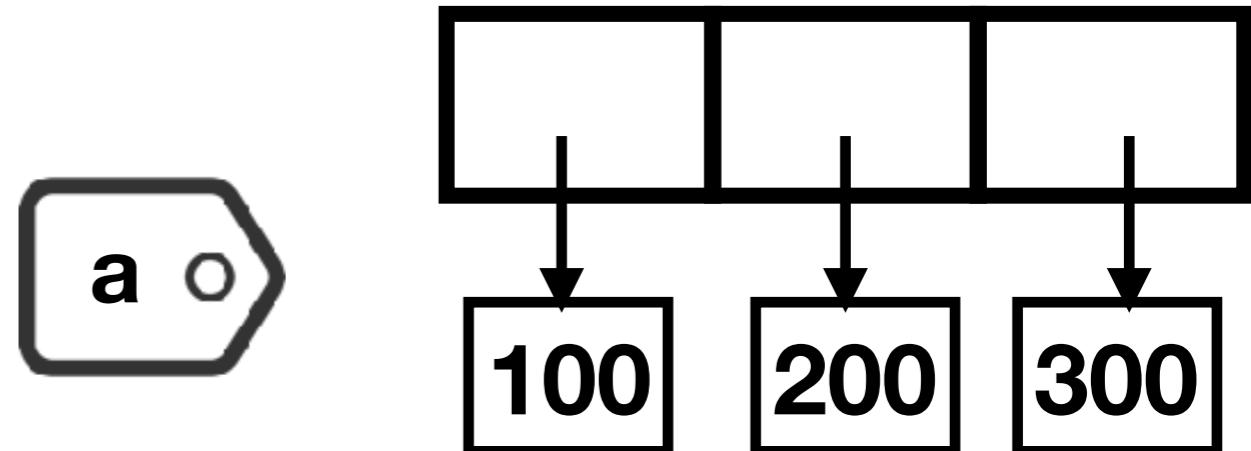
배열 원소의 크기는 int 형으로 고정됨

Python

리스트

리스트 객체 생성

```
a = [100, 200, 300]
```



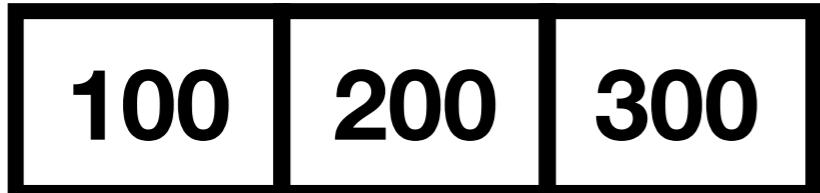
C

배열

// 배열의 선언과 초기화

```
int a[3] = {100, 200, 300};
```

a



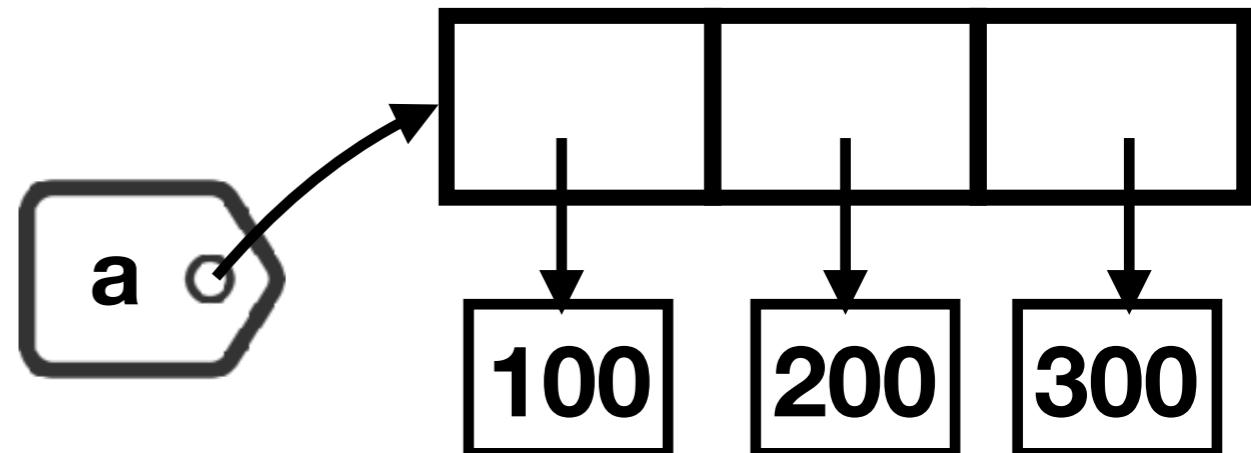
배열 원소의 크기는 int 형으로 고정됨

Python

리스트

리스트 객체 생성

```
a = [100, 200, 300]
```

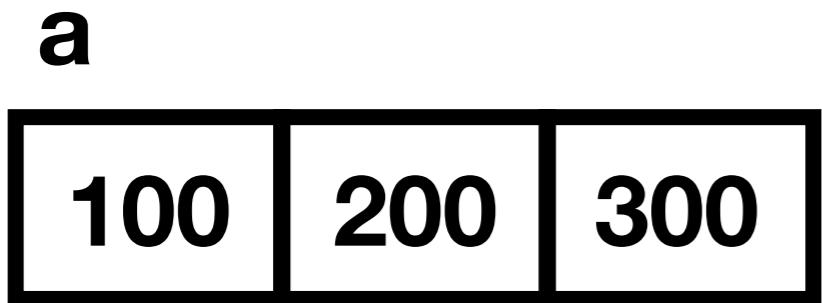


C

배열

// 배열의 선언과 초기화

```
int a[3] = {100, 200, 300};
```



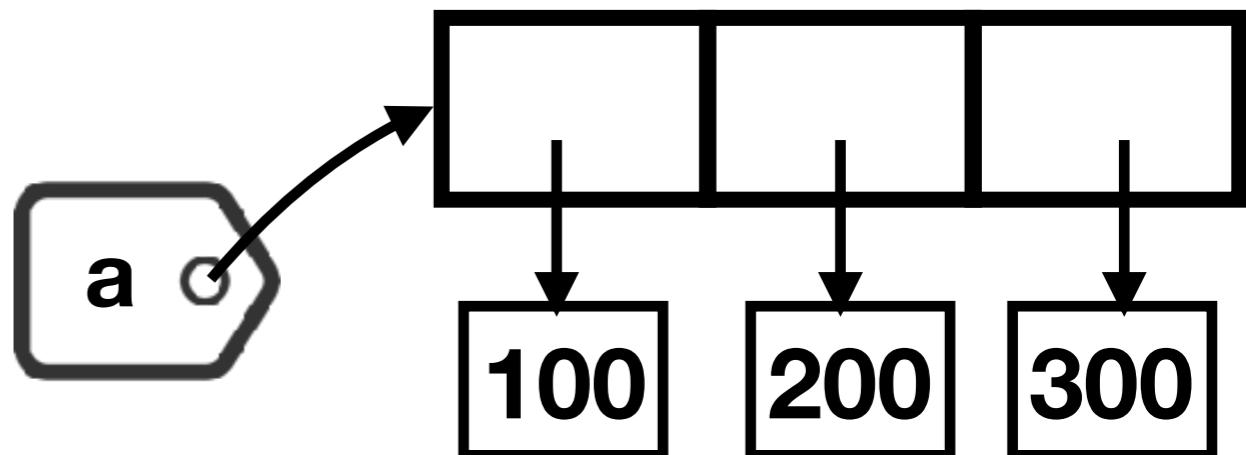
배열 원소의 크기는 int 형으로 고정됨

Python

리스트

리스트 객체 생성

```
a = [100, 200, 300]
```



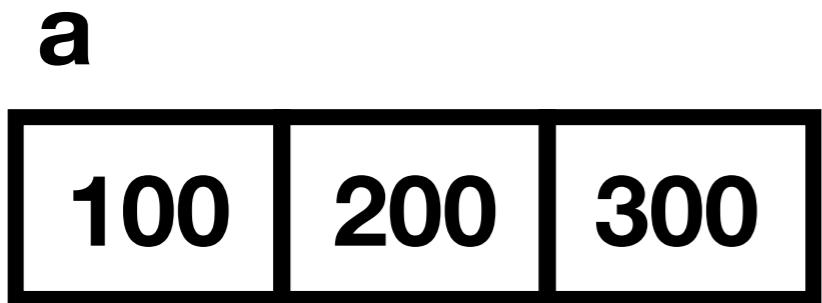
리스트의 항목은 서로 다른 자료형도 가능
리스트 요소 a[0], a[1], a[2]는 참조형임

C

배열

// 배열의 선언과 초기화

```
int a[3] = {100, 200, 300};
```



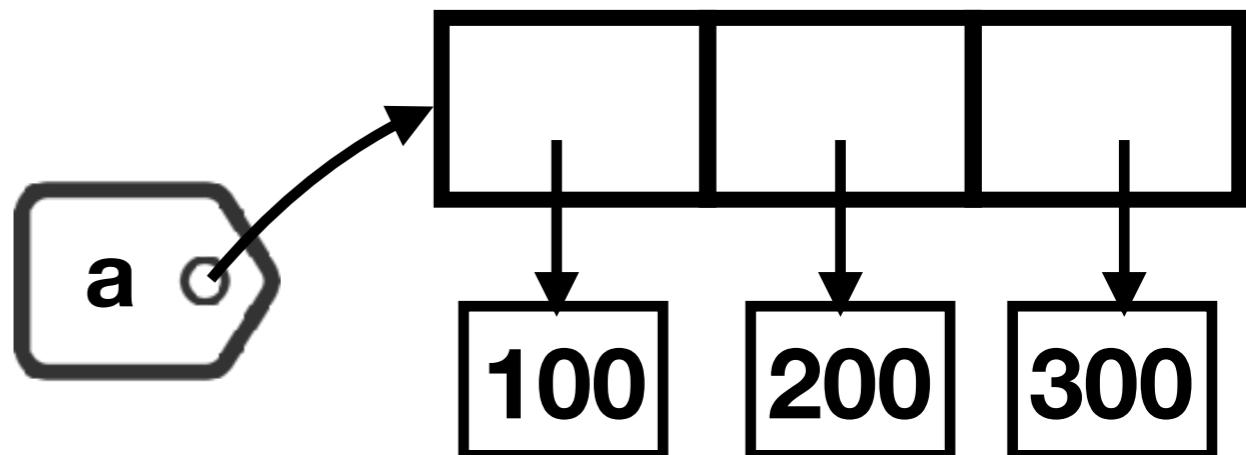
배열 원소의 크기는 int 형으로 고정됨

Python

리스트

리스트 객체 생성

```
a = [100, 200, 300]
```



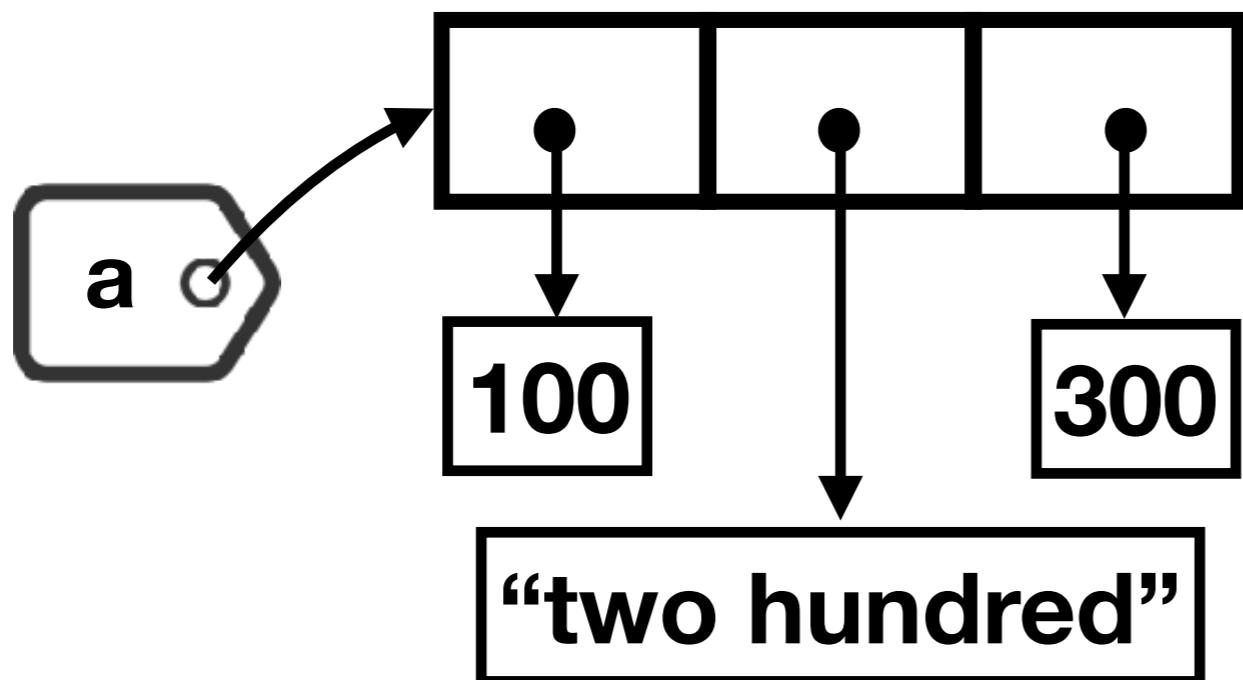
리스트의 항목은 서로 다른 자료형도 가능
리스트 요소 a[0], a[1], a[2]는 참조형임

elements are references

리스트 요소는 참조형이다!!

리스트 객체 생성

a = [100, “two hundred”, 300]



```
>>> print(a[0])  
100  
>>> print(a[1])  
two hundred  
>>> id(a)  
4512940232  
>>> id(a[0])  
4509016272  
>>> id(a[1])  
4512941744
```

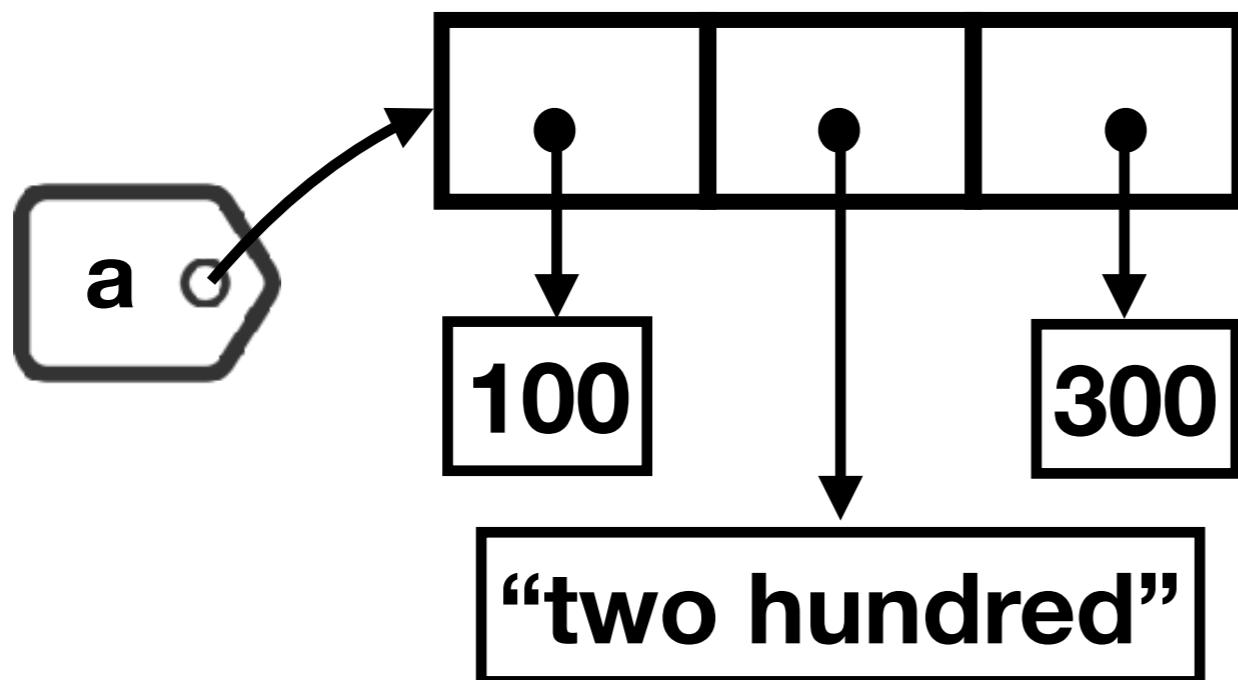
elements are references

리스트 요소는 참조형이다!!

리스트 객체 생성

```
a = [100, "two hundred", 300]
```

```
print(a[0])
```



```
>>> print(a[0])
100
>>> print(a[1])
two hundred
>>> id(a)
4512940232
>>> id(a[0])
4509016272
>>> id(a[1])
4512941744
```

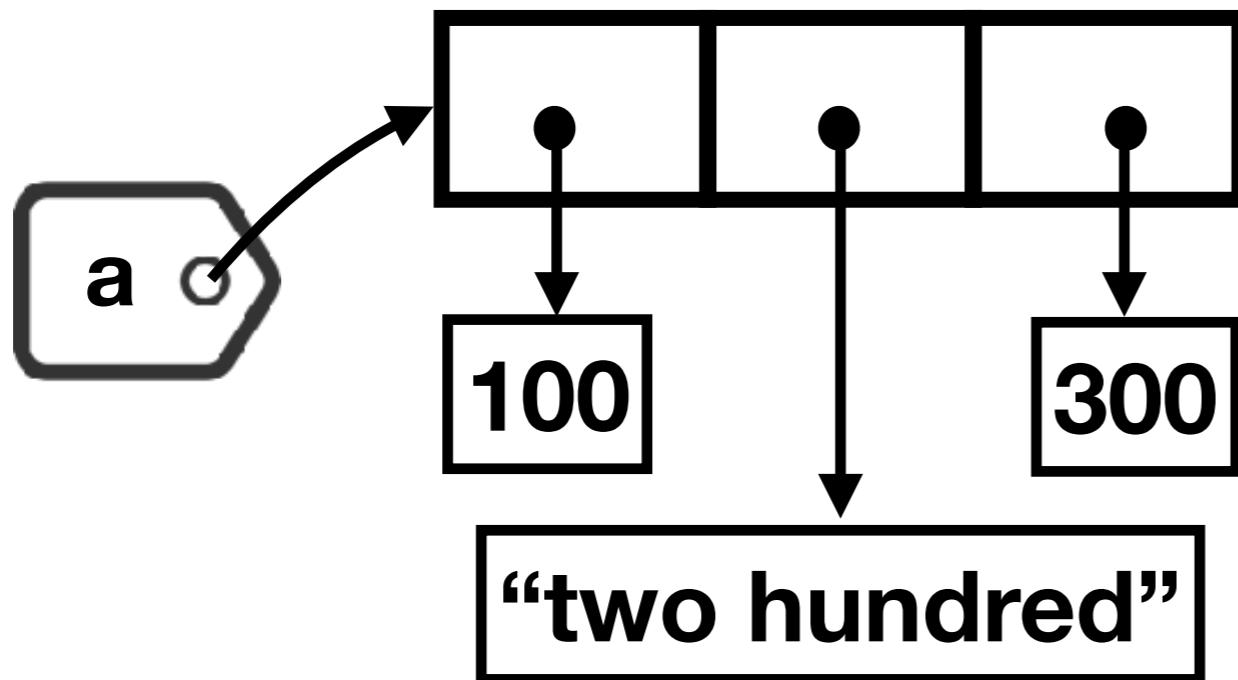
elements are references

리스트 요소는 참조형이다!!

리스트 객체 생성

```
a = [100, "two hundred", 300]
```

```
print(a[0])
```



```
>>> print(a[0])
100
>>> print(a[1])
two hundred
>>> id(a)
4512940232
>>> id(a[0])
4509016272
>>> id(a[1])
4512941744
```

elements are references

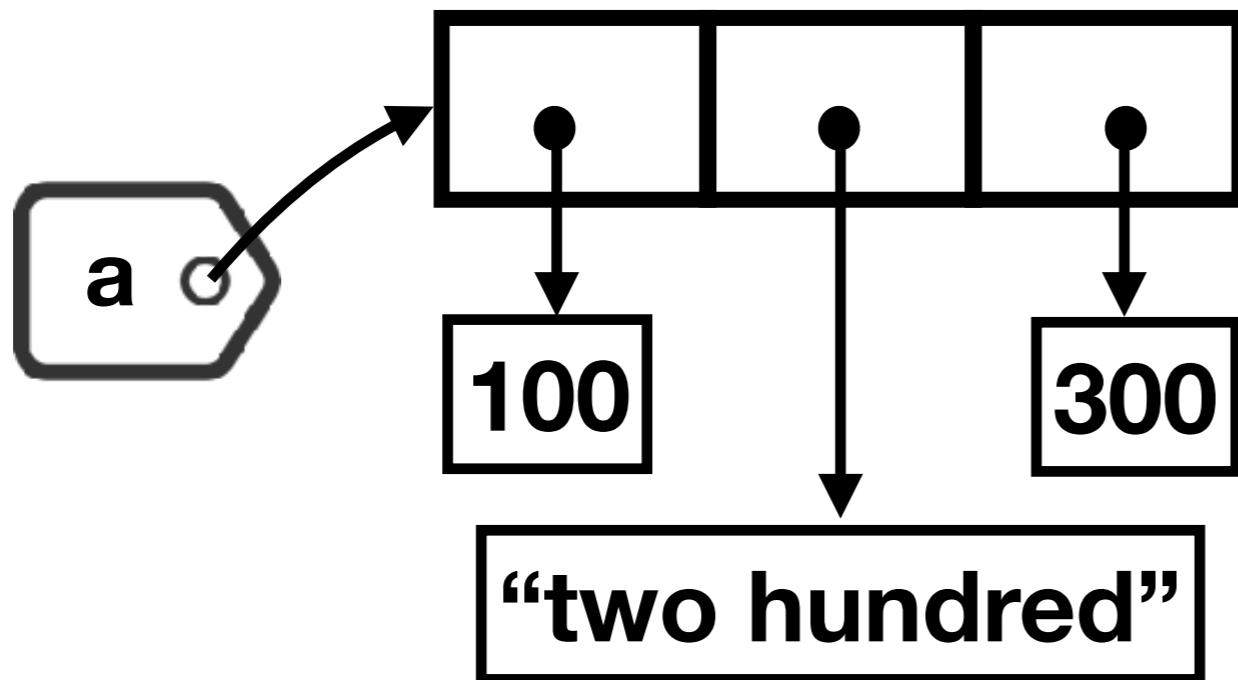
리스트 요소는 참조형이다!!

리스트 객체 생성

```
a = [100, "two hundred", 300]
```

```
print(a[0])
```

```
print(a[1])
```



```
>>> print(a[0])
100
>>> print(a[1])
two hundred
>>> id(a)
4512940232
>>> id(a[0])
4509016272
>>> id(a[1])
4512941744
```

elements are references

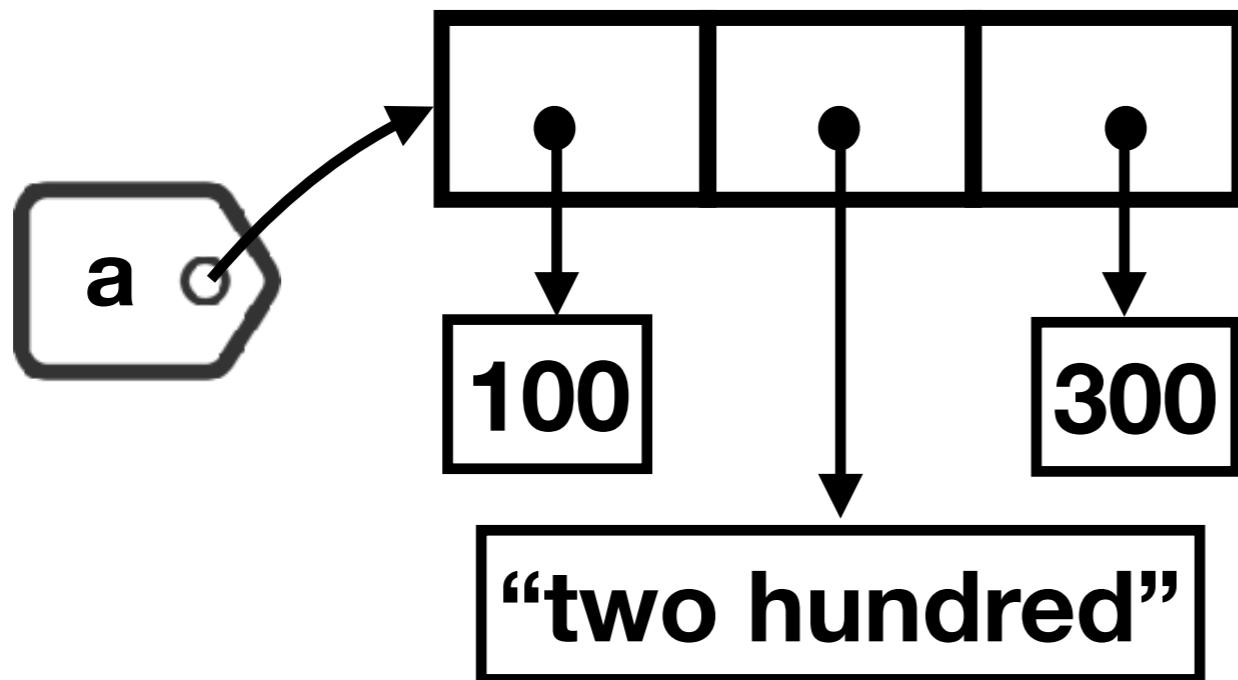
리스트 요소는 참조형이다!!

리스트 객체 생성

```
a = [100, "two hundred", 300]
```

```
print(a[0])
```

```
print(a[1])
```

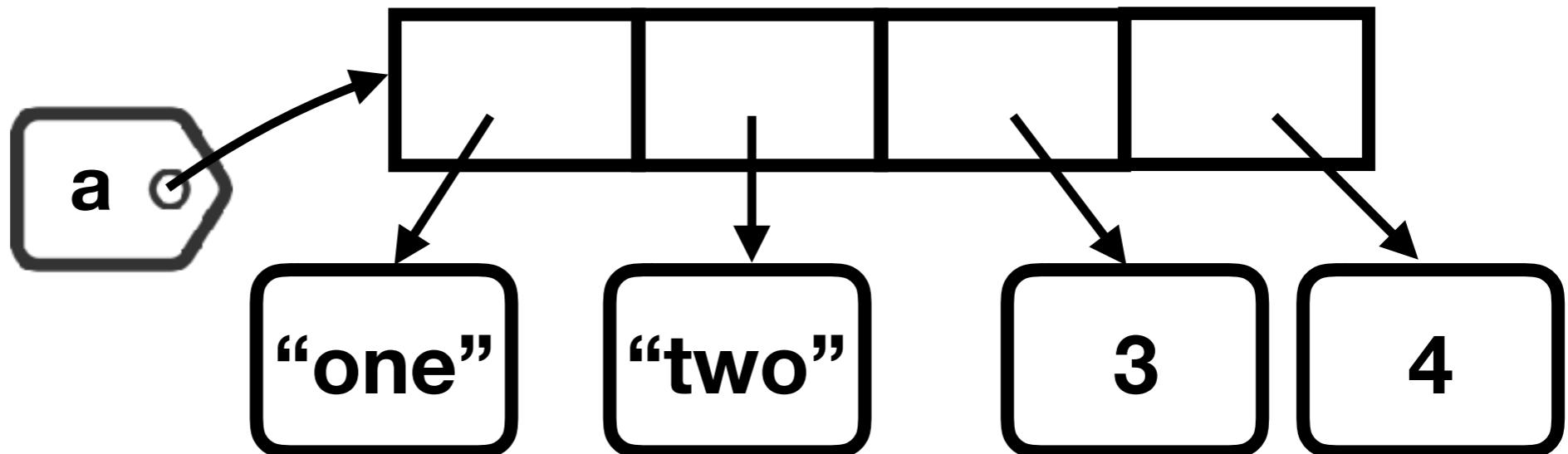


```
>>> print(a[0])
100
>>> print(a[1])
two hundred
>>> id(a)
4512940232
>>> id(a[0])
4509016272
>>> id(a[1])
4512941744
```

리스트 요소의 재할당

리스트 객체 생성

```
a = ["one", "two", 3, 4]
```



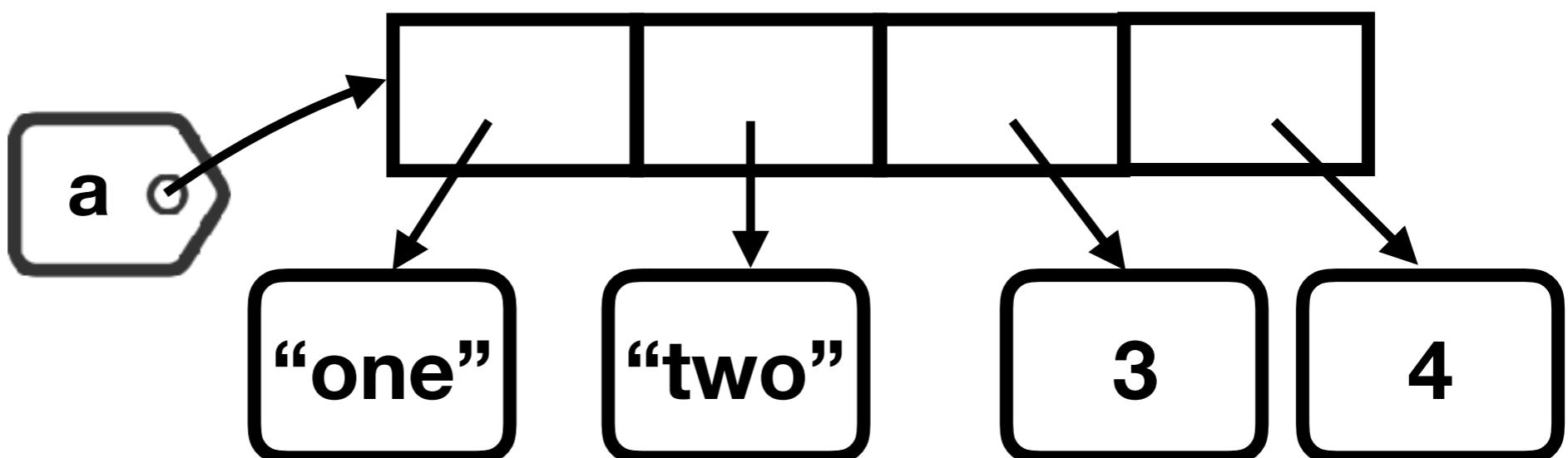
리스트 요소의 재할당

리스트 객체 생성

```
a = ["one", "two", 3, 4]
```

리스트 요소의 재할당

```
a[1] = 2
```



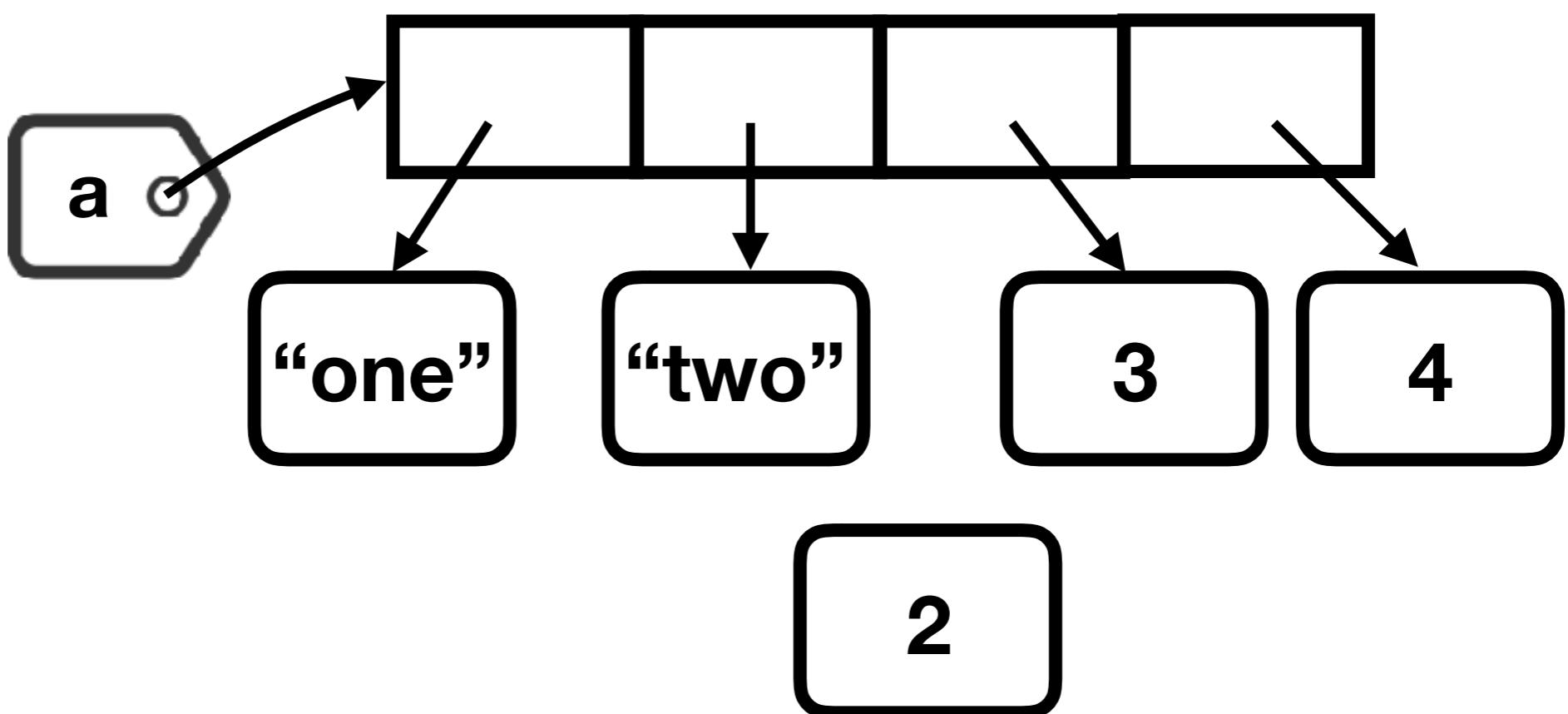
리스트 요소의 재할당

리스트 객체 생성

```
a = ["one", "two", 3, 4]
```

리스트 요소의 재할당

```
a[1] = 2
```



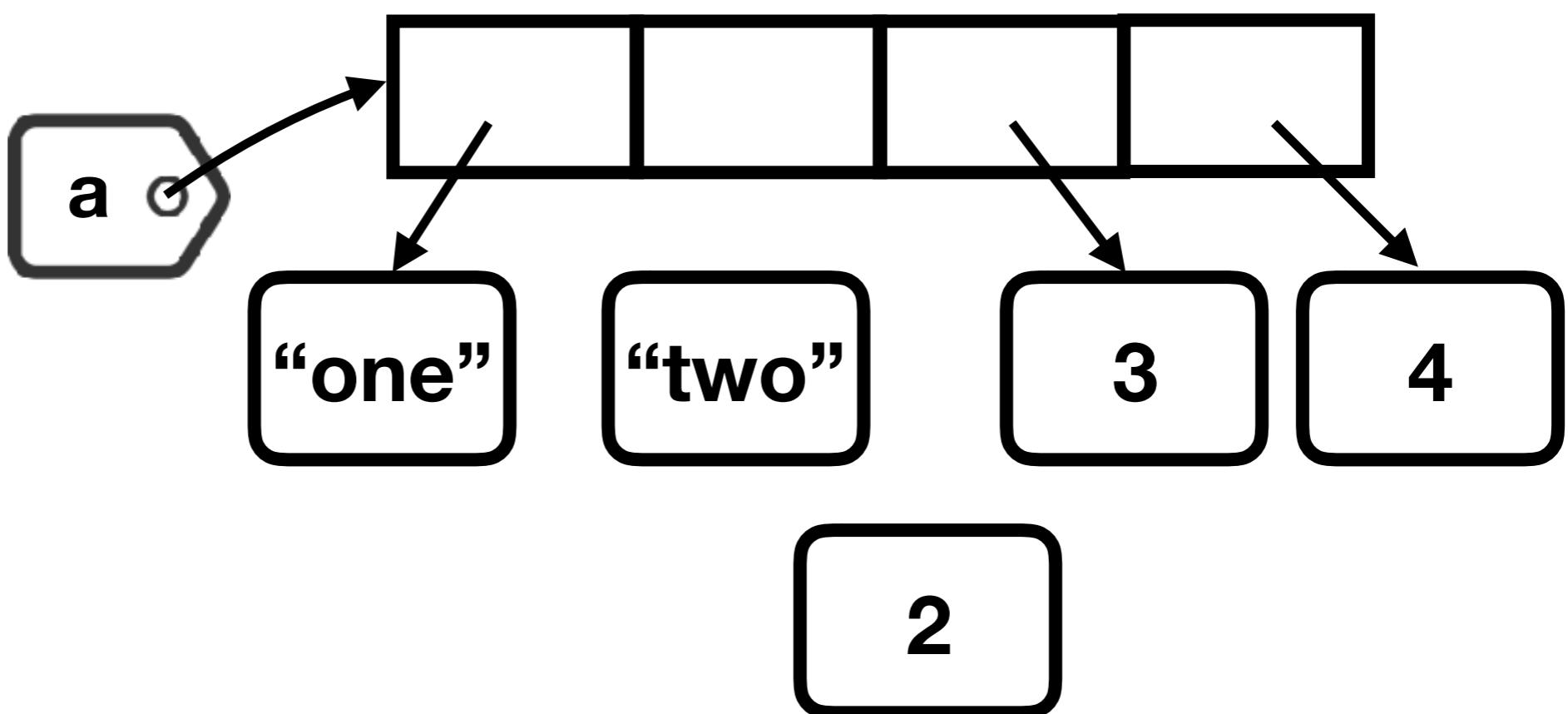
리스트 요소의 재할당

리스트 객체 생성

```
a = ["one", "two", 3, 4]
```

리스트 요소의 재할당

```
a[1] = 2
```



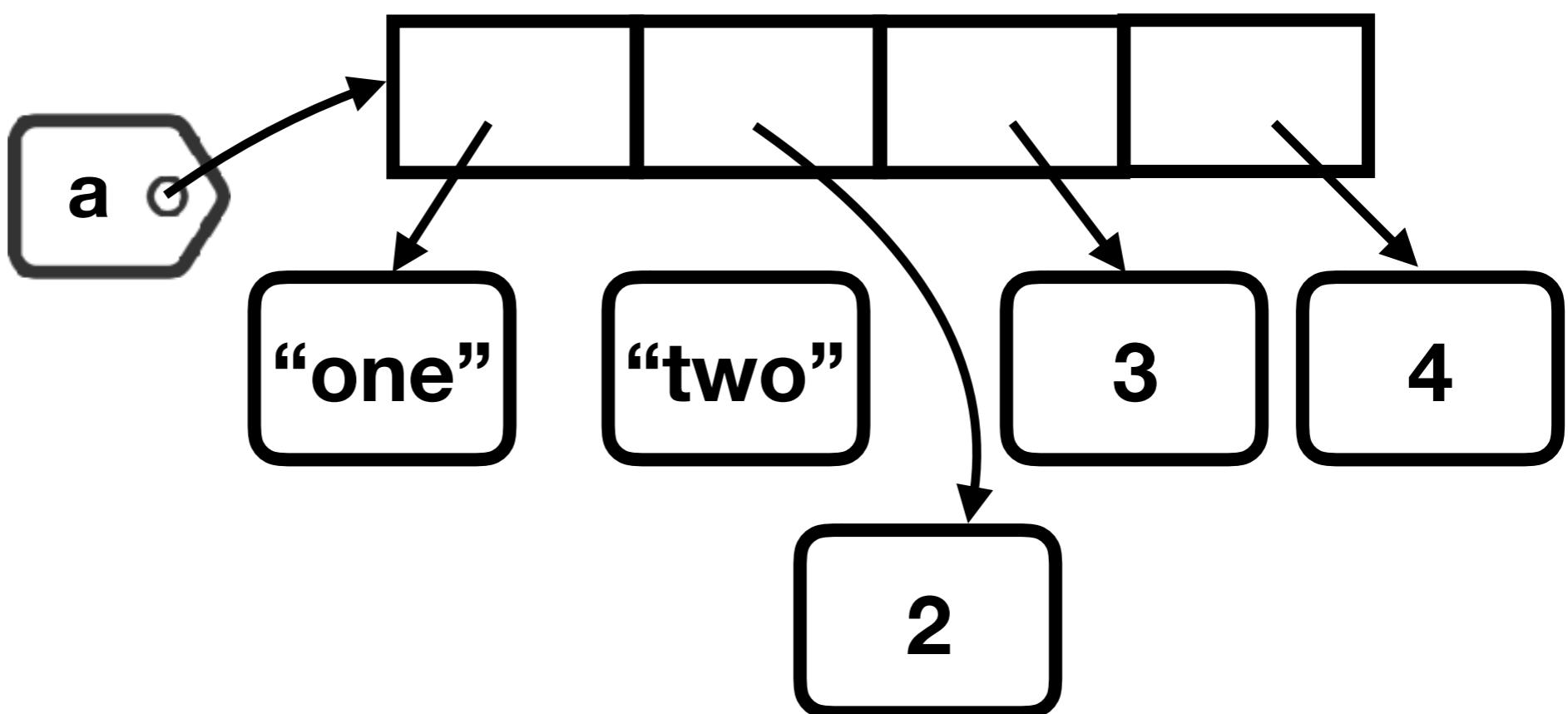
리스트 요소의 재할당

리스트 객체 생성

```
a = ["one", "two", 3, 4]
```

리스트 요소의 재할당

```
a[1] = 2
```



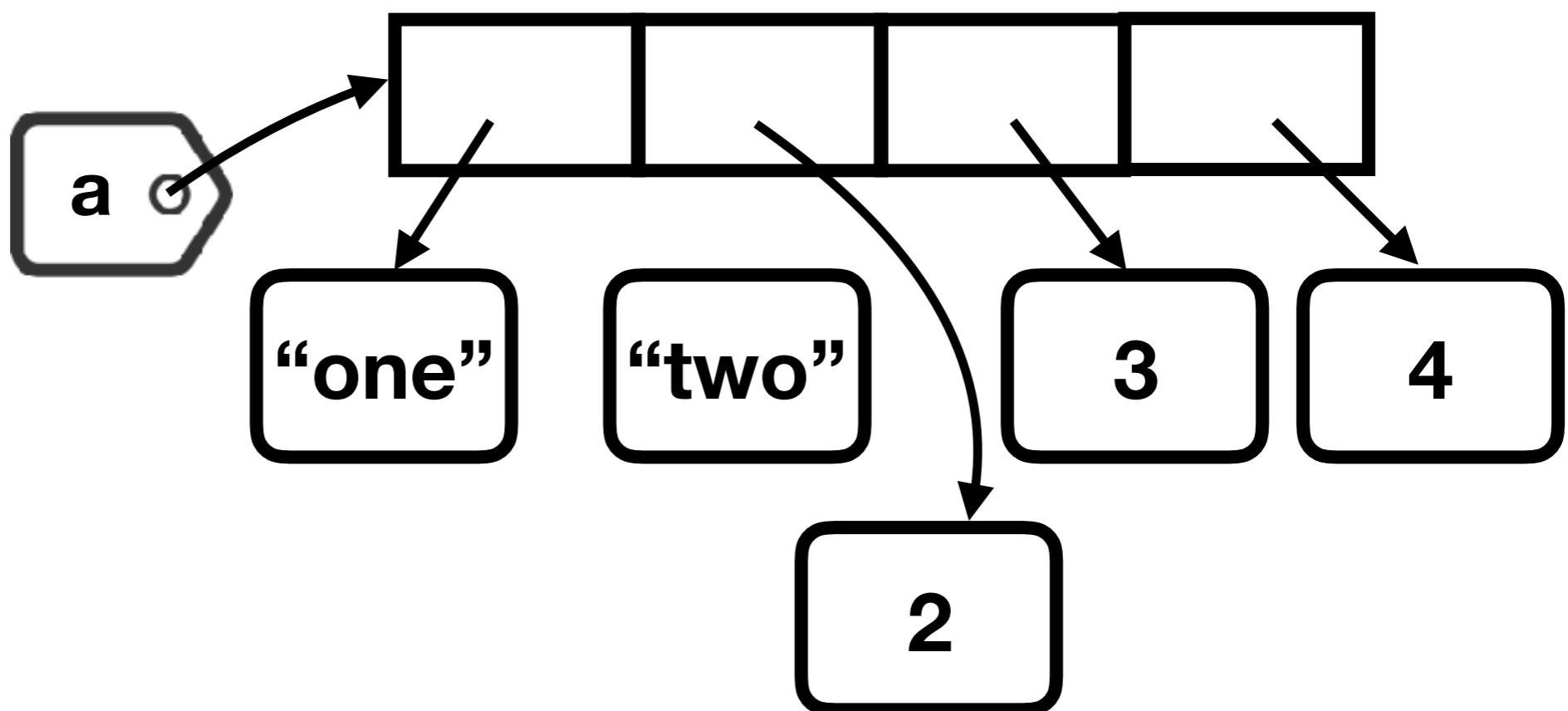
리스트 요소의 재할당

리스트 객체 생성

```
a = ["one", "two", 3, 4]
```

리스트 요소의 재할당

```
a[1] = 2
```



유연한 데이터 처리가 가능한 반면(만능 가제트 형사) 속도 저하의 원인

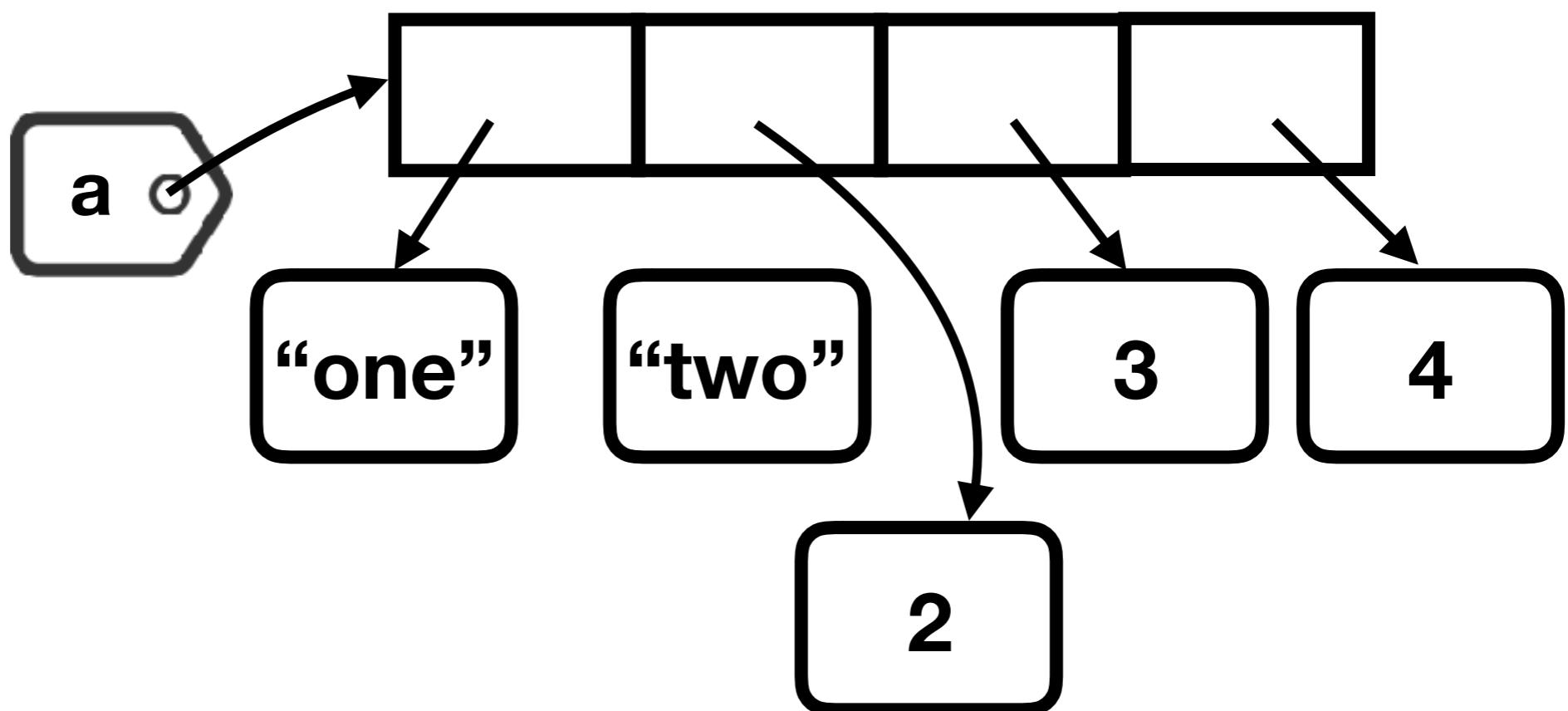
리스트 요소의 재할당

리스트 객체 생성

```
a = ["one", "two", 3, 4]
```

리스트 요소의 재할당

```
a[1] = 2
```



유연한 데이터 처리가 가능한 반면(만능 객체 형사) 속도 저하의 원인

리스트 연산의 비교

```
# 리스트 객체 생성  
a = [1, 2, 3, 4]  
a.append(5)      # 원소의 추가  
print(a)
```

리스트 연산의 비교

```
# 리스트 객체 생성  
a = [1, 2, 3, 4]  
a.append(5)      # 원소의 추가  
print(a)
```

```
# 리스트 객체 생성
```

리스트 연산의 비교

```
# 리스트 객체 생성  
a = [1, 2, 3, 4]  
a.append(5)      # 원소의 추가  
print(a)
```

```
# 리스트 객체 생성  
a = [1, 2, 3, 4]
```

리스트 연산의 비교

```
# 리스트 객체 생성
```

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

```
a.append(5)      # 원소의 추가
```

```
print(a)
```

```
# 리스트 객체 생성
```

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

```
a = a+[5]      # 원소의 추가
```

리스트 연산의 비교

```
# 리스트 객체 생성
```

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

```
a.append(5)      # 원소의 추가
```

```
print(a)
```

```
# 리스트 객체 생성
```

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

```
a = a+[5]        # 원소의 추가
```

```
print(a)
```

리스트 연산의 비교

```
# 리스트 객체 생성
```

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

```
a.append(5)      # 원소의 추가
```

```
print(a)
```

```
# 리스트 객체 생성
```

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

```
a = a+[5]        # 원소의 추가
```

```
print(a)
```

결과

리스트 연산의 비교

```
# 리스트 객체 생성
```

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

```
a.append(5)      # 원소의 추가
```

```
print(a)
```

```
# 리스트 객체 생성
```

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

```
a = a+[5]        # 원소의 추가
```

```
print(a)
```

결과

[1, 2, 3, 4, 5]

a.append(5) # 원소의 추가

a.append(5) # 원소의 추가

vs

a.append(5) # 원소의 추가

vs

a = a+[5] # 원소의 추가

a.append(5) # 원소의 추가

vs

a = a+[5] # 원소의 추가

어떤 차이가 있을까요?

a.append(5) # 원소의 추가

vs

a = a+[5] # 원소의 추가

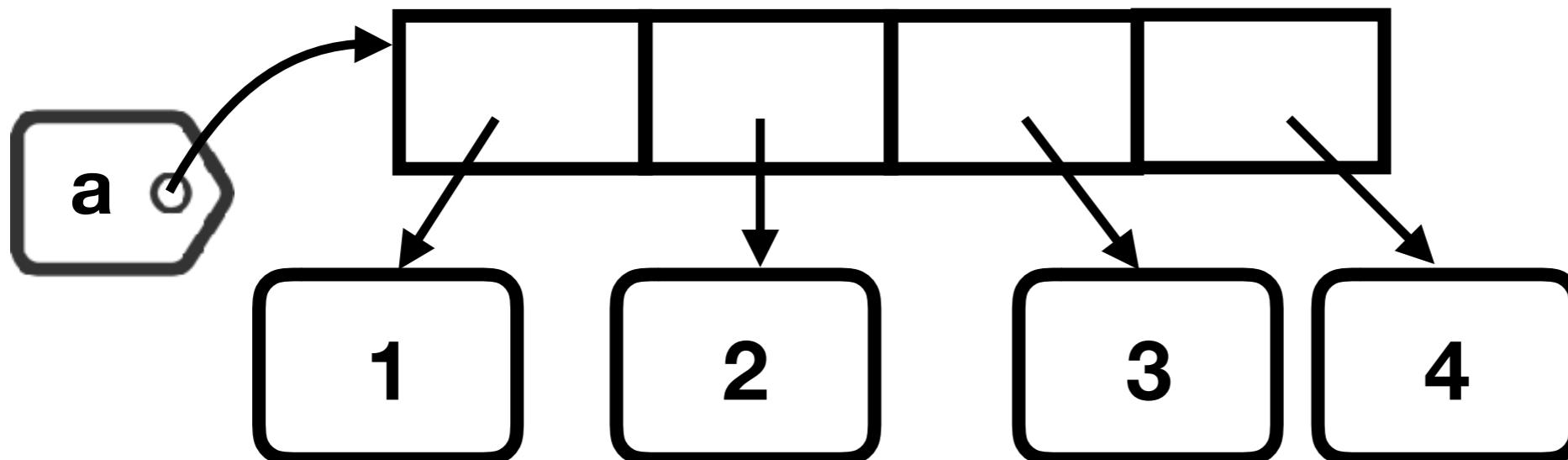
어떤 차이가 있을까요?

결과는 동일하지만 수행과정은 큰 차이가 있음

리스트 요소의 추가

리스트 객체 생성

a = [1, 2, 3, 4]

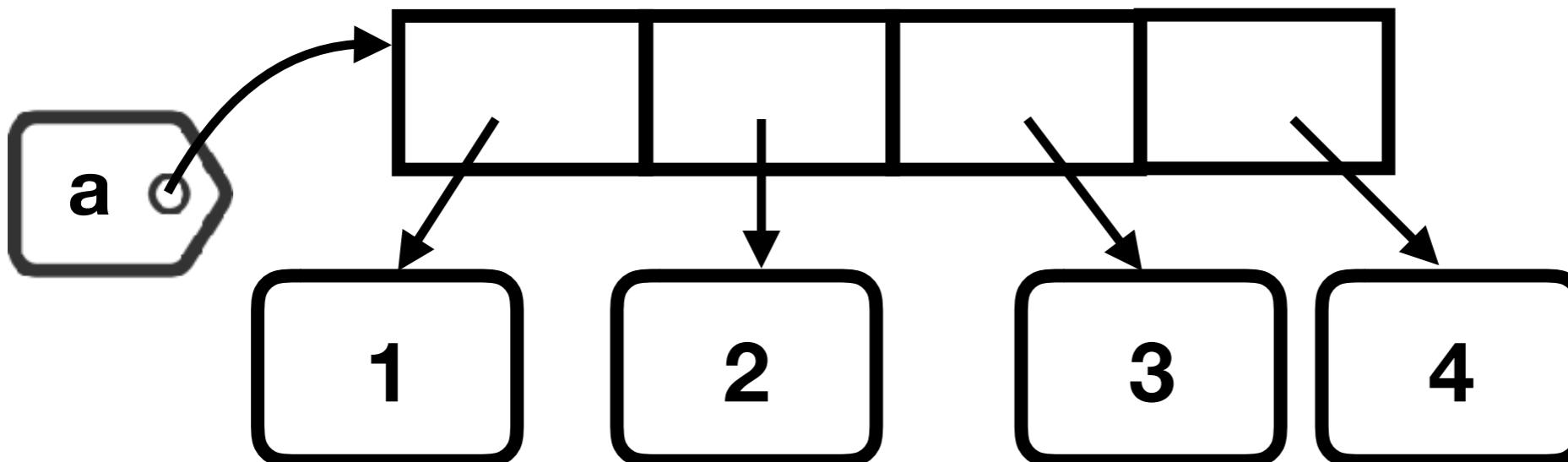


리스트 요소의 추가

리스트 객체 생성

a = [1, 2, 3, 4]

a.append(5) # 리스트 객체의 변경(mutating)

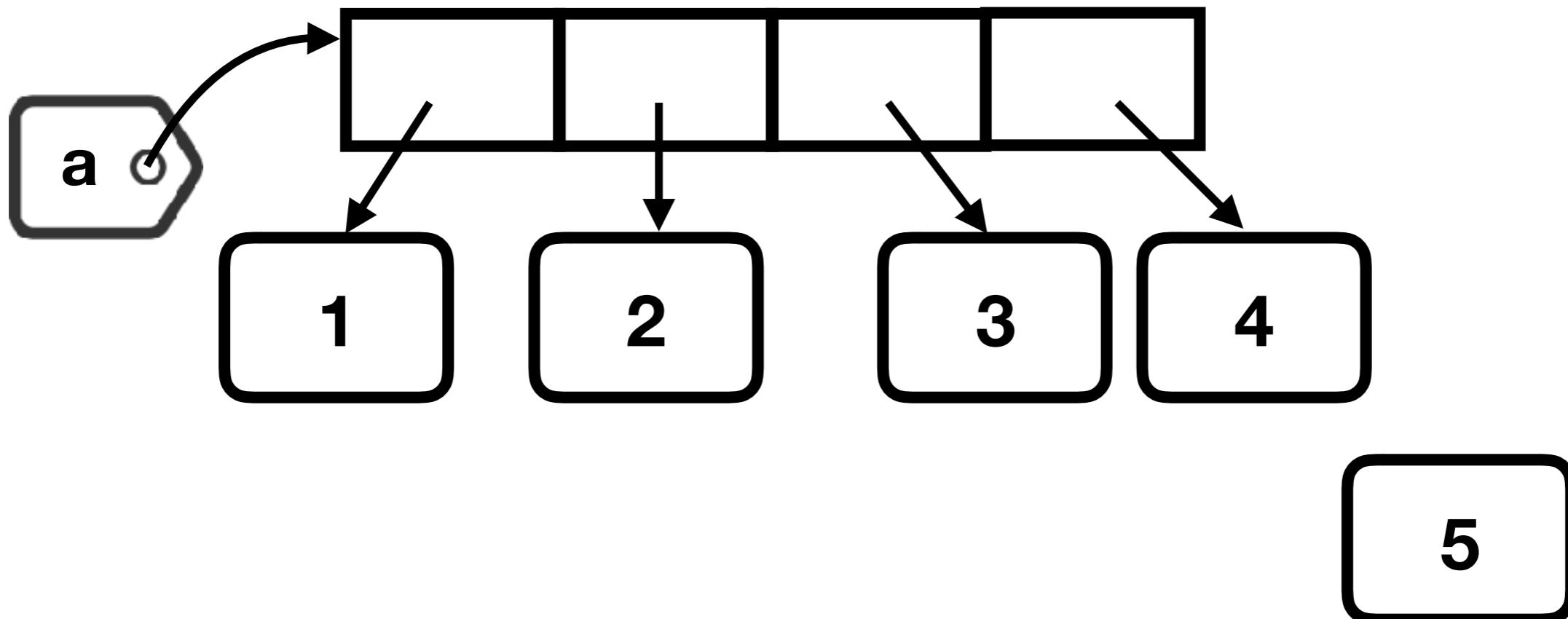


리스트 요소의 추가

리스트 객체 생성

a = [1, 2, 3, 4]

a.append(5) # 리스트 객체의 변경(mutating)

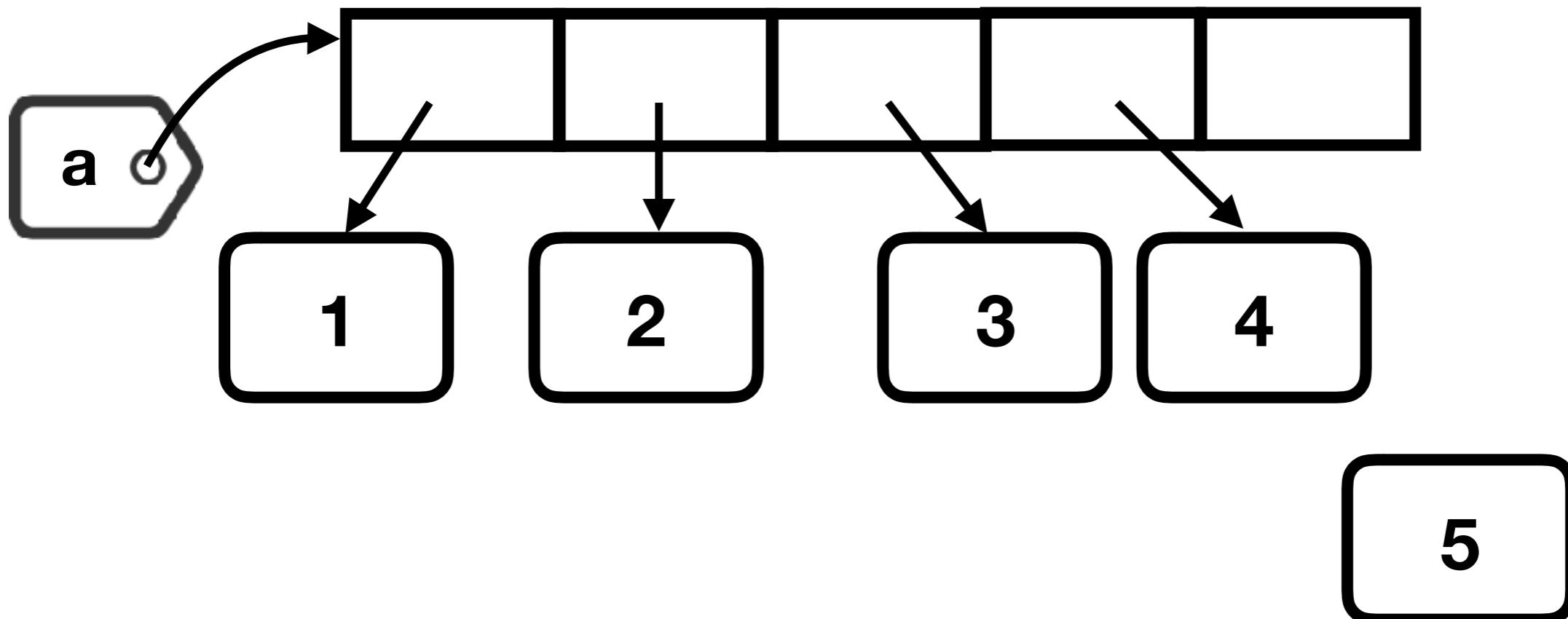


리스트 요소의 추가

리스트 객체 생성

a = [1, 2, 3, 4]

a.append(5) # 리스트 객체의 변경(mutating)

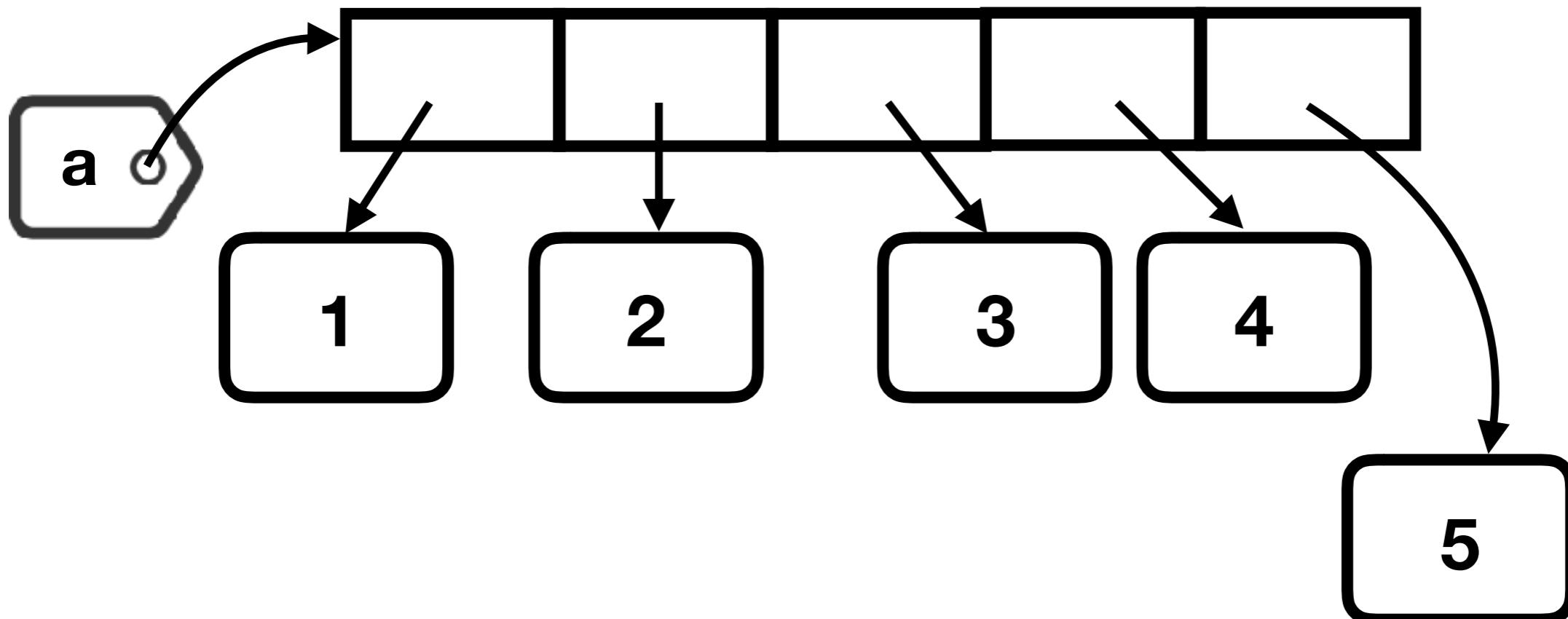


리스트 요소의 추가

리스트 객체 생성

a = [1, 2, 3, 4]

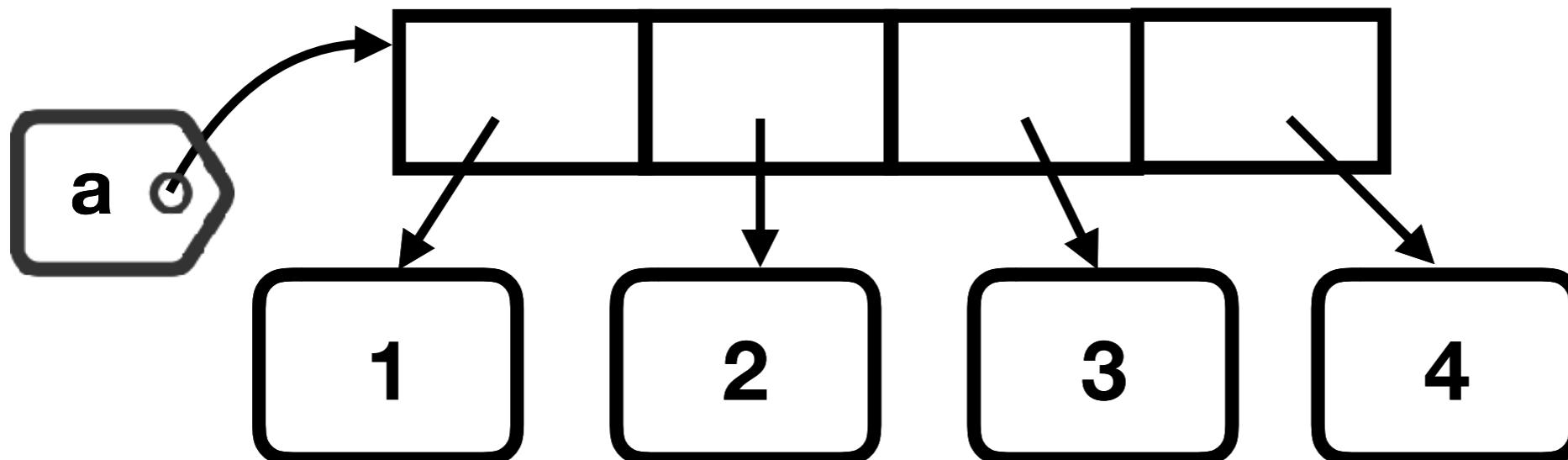
a.append(5) # 리스트 객체의 변경(mutating)



리스트의 덧셈과 재할당

리스트 객체 생성

a = [1, 2, 3, 4]

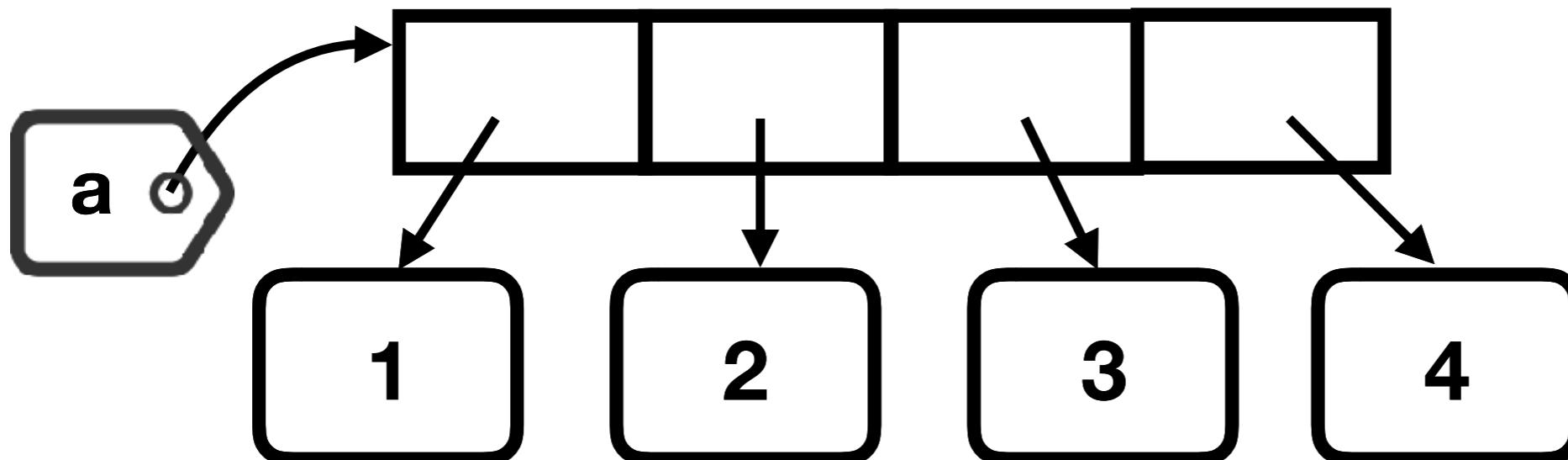


리스트의 덧셈과 재할당

리스트 객체 생성

a = [1, 2, 3, 4]

a = a + [5] # 리스트 객체의 재바인딩(rebinding)

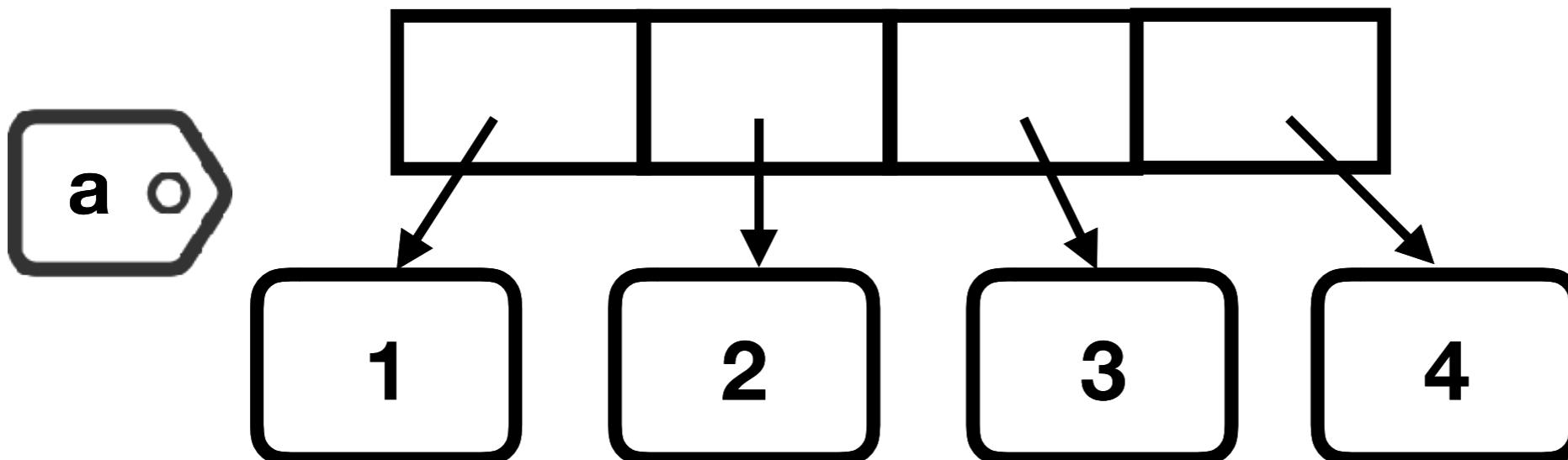


리스트의 덧셈과 재할당

리스트 객체 생성

a = [1, 2, 3, 4]

a = a + [5] # 리스트 객체의 재바인딩(rebinding)

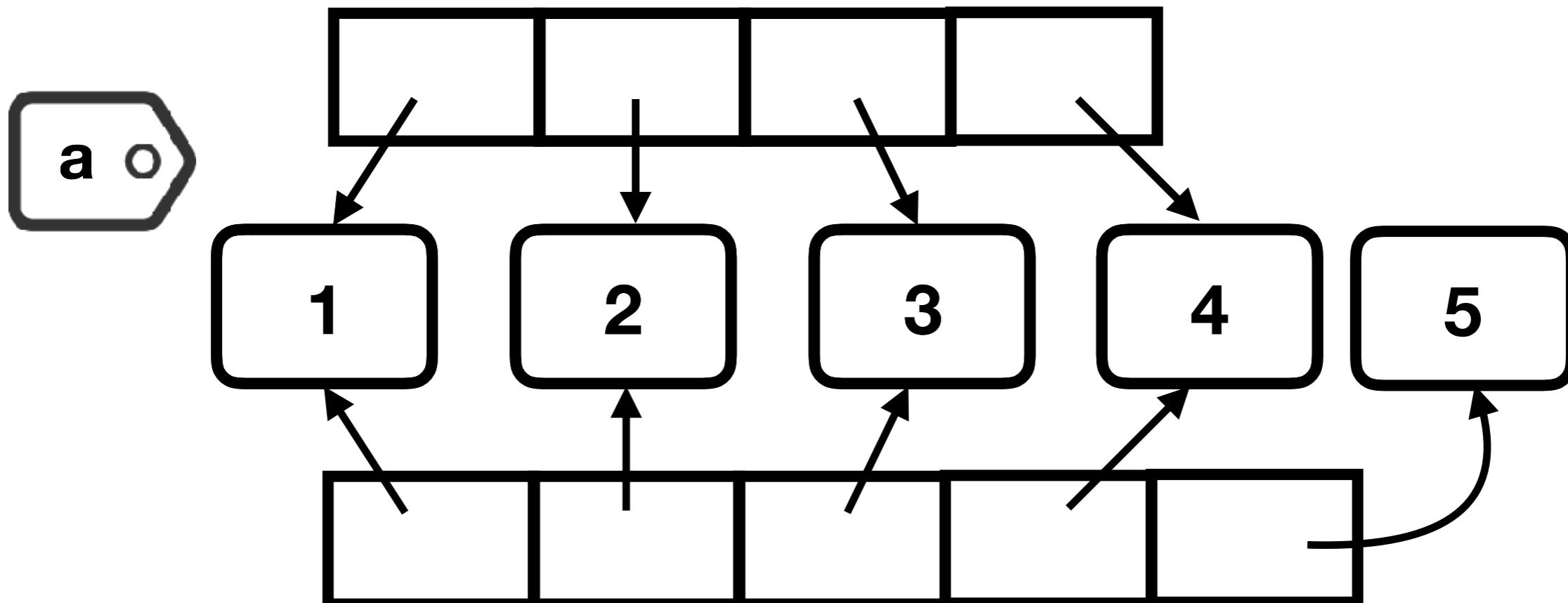


리스트의 덧셈과 재할당

리스트 객체 생성

a = [1, 2, 3, 4]

a = a + [5] # 리스트 객체의 재바인딩(rebinding)

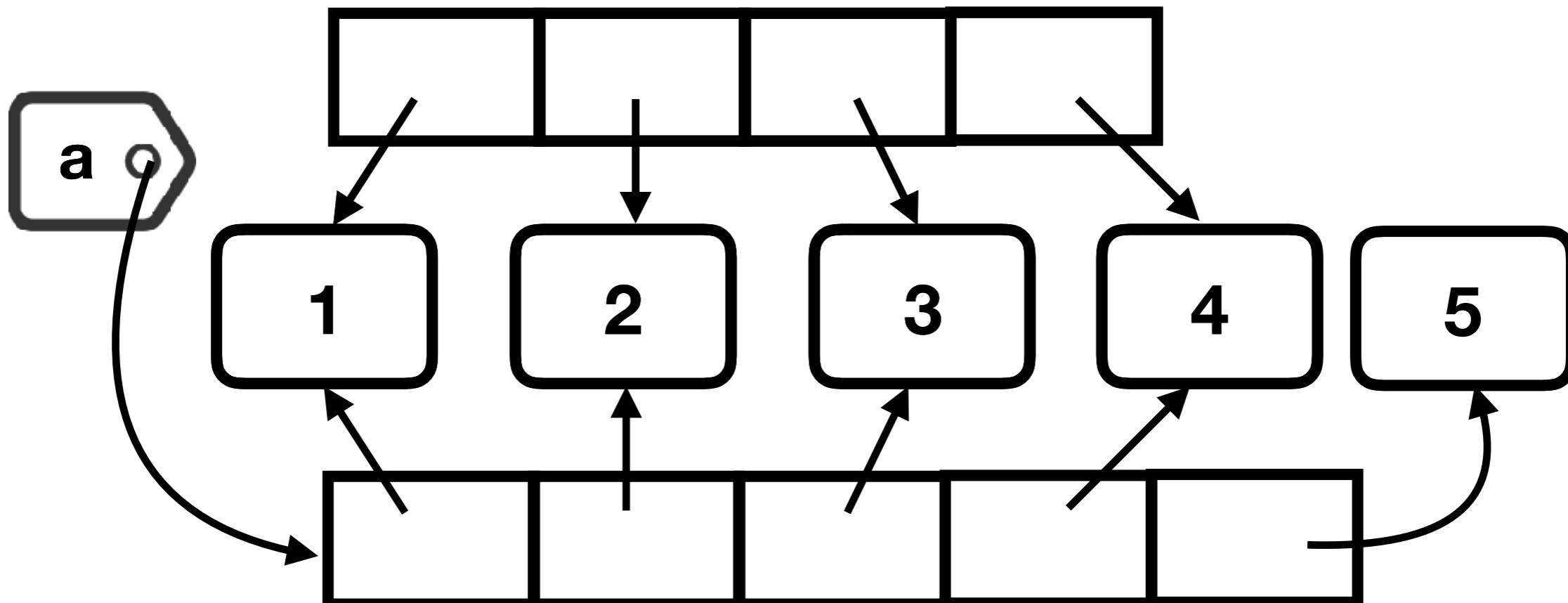


리스트의 덧셈과 재할당

리스트 객체 생성

a = [1, 2, 3, 4]

a = a + [5] # 리스트 객체의 재바인딩(rebinding)

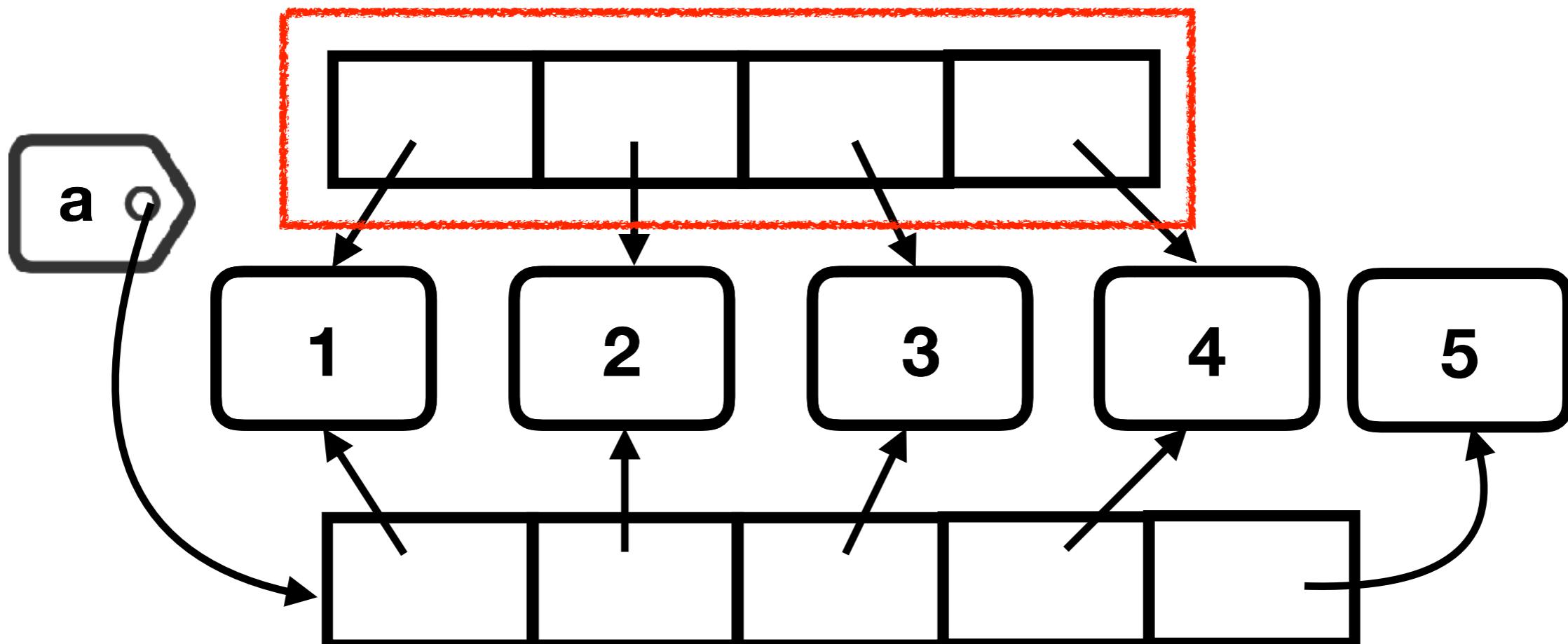


리스트의 덧셈과 재할당

리스트 객체 생성

a = [1, 2, 3, 4]

a = a + [5] # 리스트 객체의 재바인딩(rebinding)

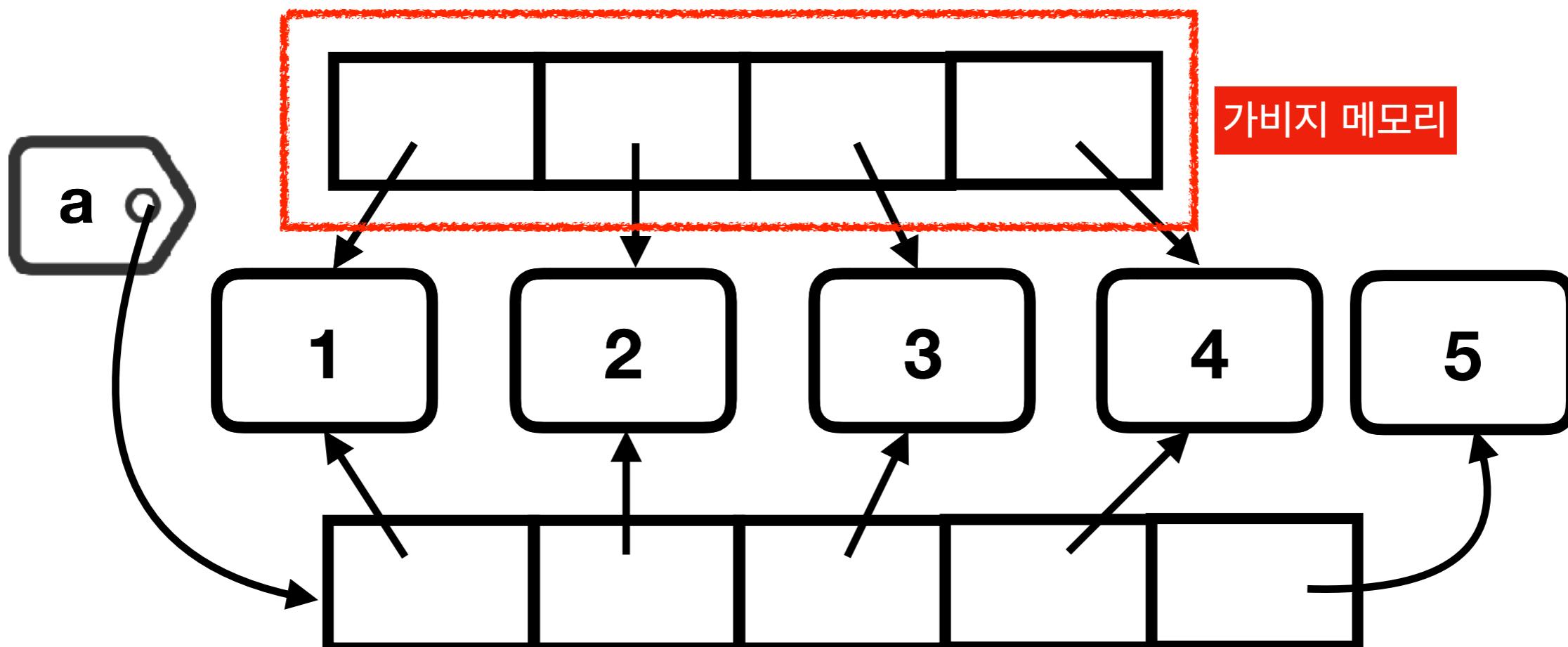


리스트의 덧셈과 재할당

리스트 객체 생성

a = [1, 2, 3, 4]

a = a + [5] # 리스트 객체의 재바인딩(rebinding)

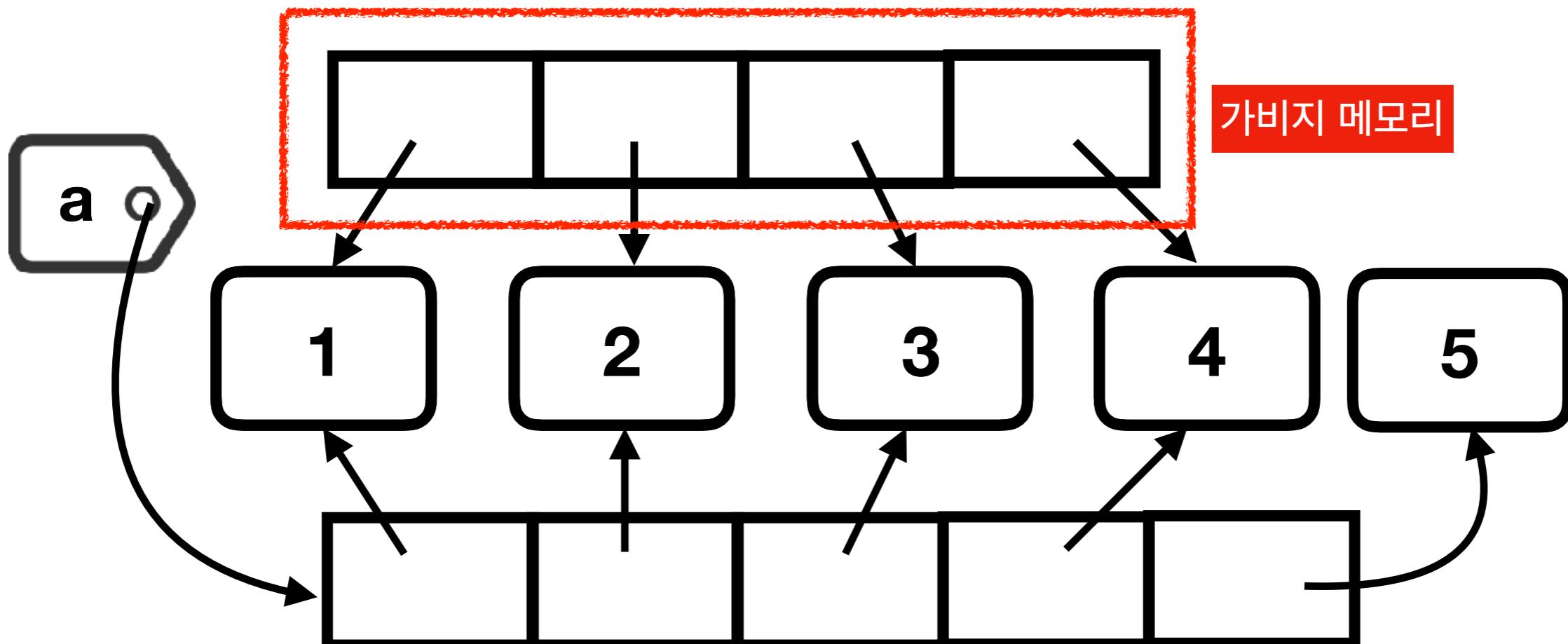


리스트의 덧셈과 재할당

리스트 객체 생성

a = [1, 2, 3, 4]

a = a + [5] # 리스트 객체의 재바인딩(rebinding)



Lab

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

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

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

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 재바인딩 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

```
[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 = 0.014089584350585938

```
[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 = 22.02604579925537

```
[6] 1 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)
```

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

↳ append(i) 소요시간 = 0.014089584350585938

```
[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 = 22.02604579925537

```
[6] 1 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)
```

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

↳ append(i) 소요시간 = 0.014089584350585938

```
[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)
```

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

↳ + [i]의 소요시간 = 22.02604579925537

```
[6] 1 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)
```

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

↳ append(i) 소요시간 = 0.014089584350585938

```
[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)
```

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

↳ + [i]의 소요시간 = 22.02604579925537

```
[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)
```

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

↳ append(i) 소요시간 = 0.014089584350585938

```
[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)
```

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

↳ + [i]의 소요시간 = 22.02604579925537

```
[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

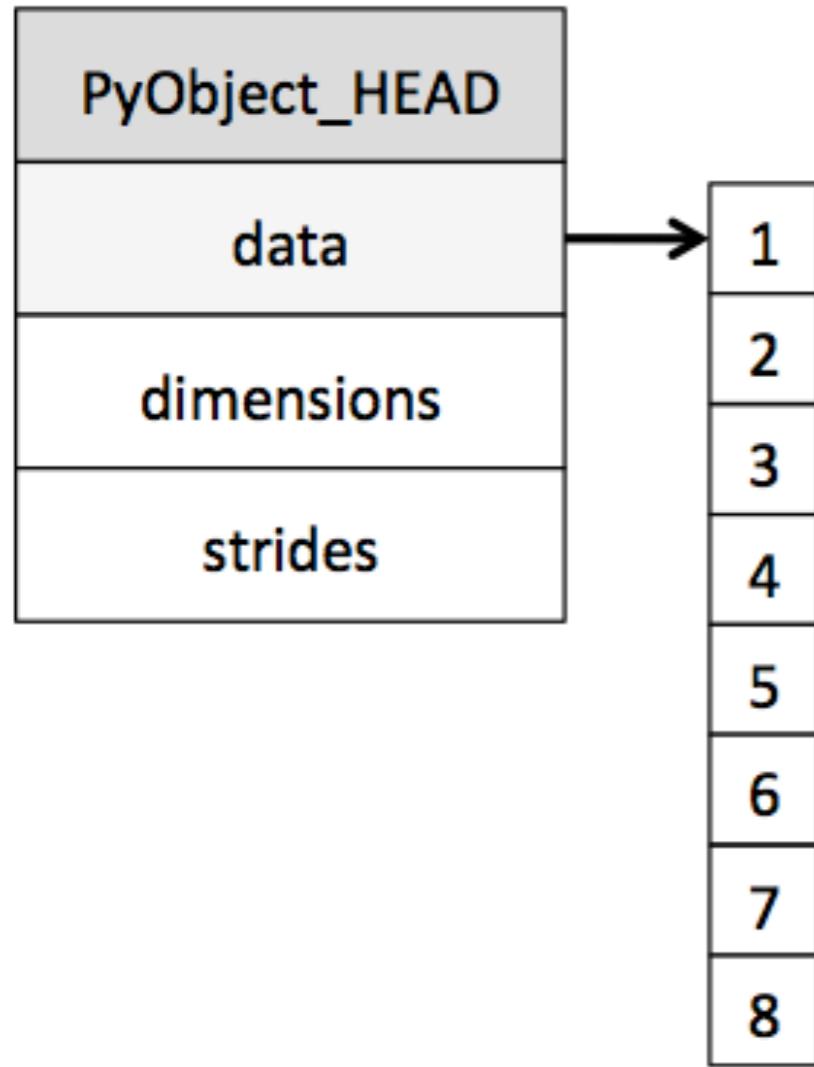
```
[15] 1 print('성능차이 {} 배'.format(t2/t4))
```

성능차이가 자그마치
34,841배

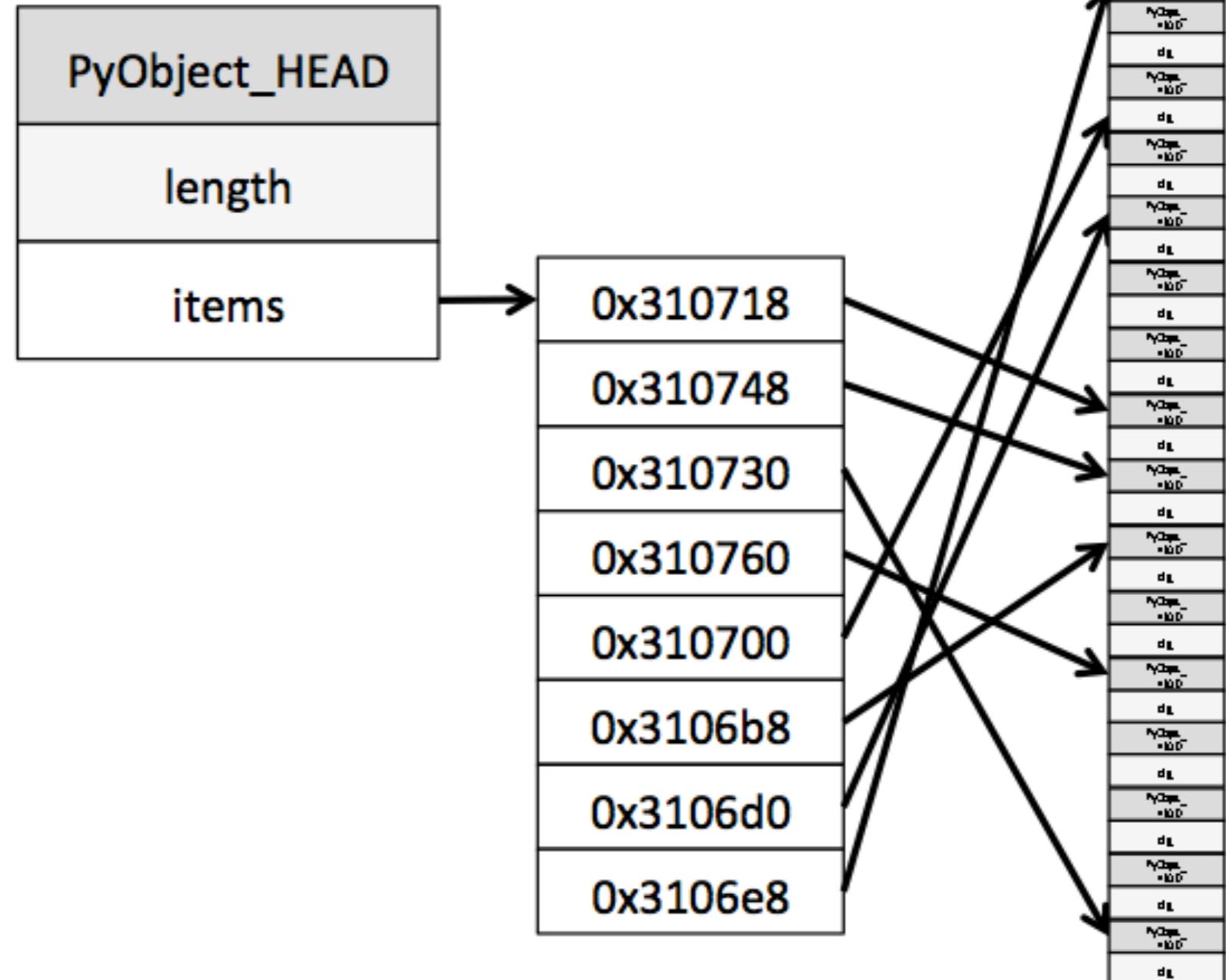
⇒ 성능차이 34841.64192017862 배

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

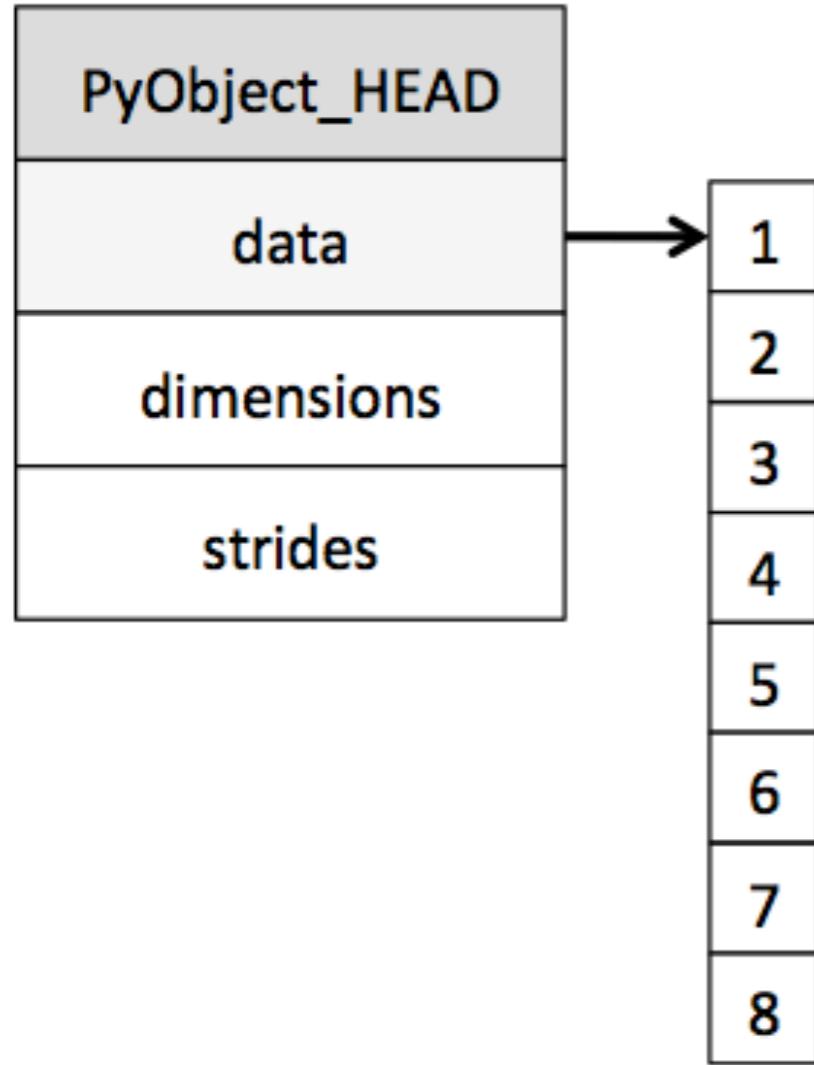
Numpy Array



Python List

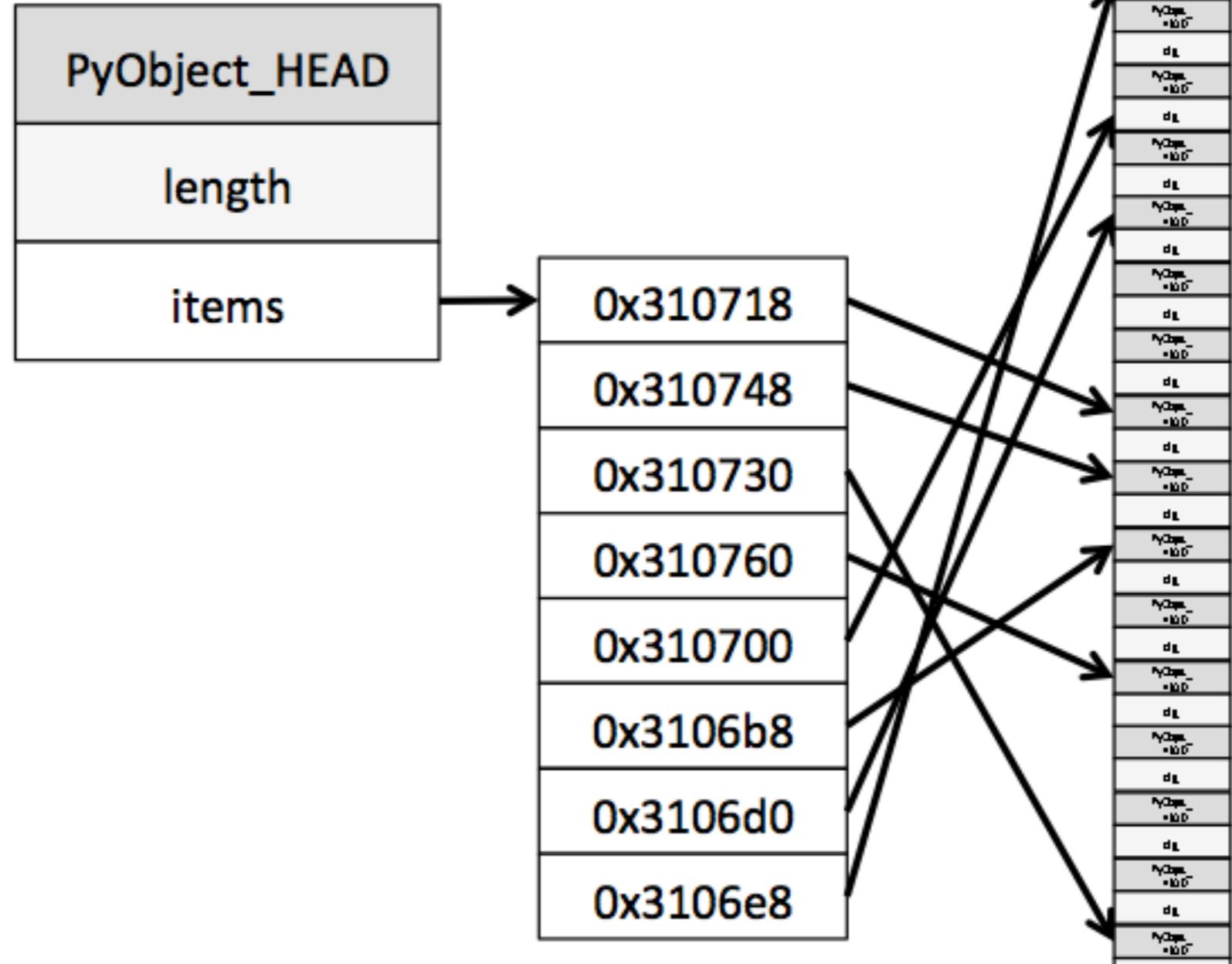


Numpy Array

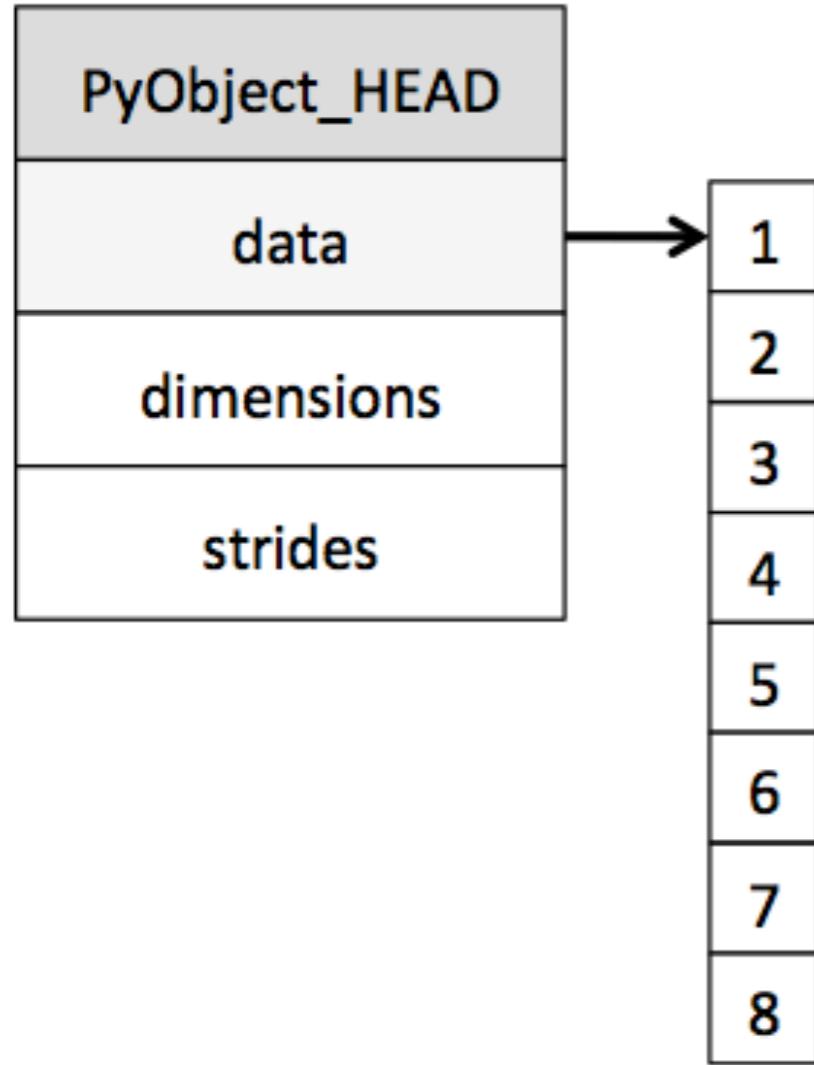


numpy의 ndarray 객체
- 동일 자료형의 객체를
- 이웃한 메모리 위치에 삽입
- C언어의 배열과 유사하다

Python List

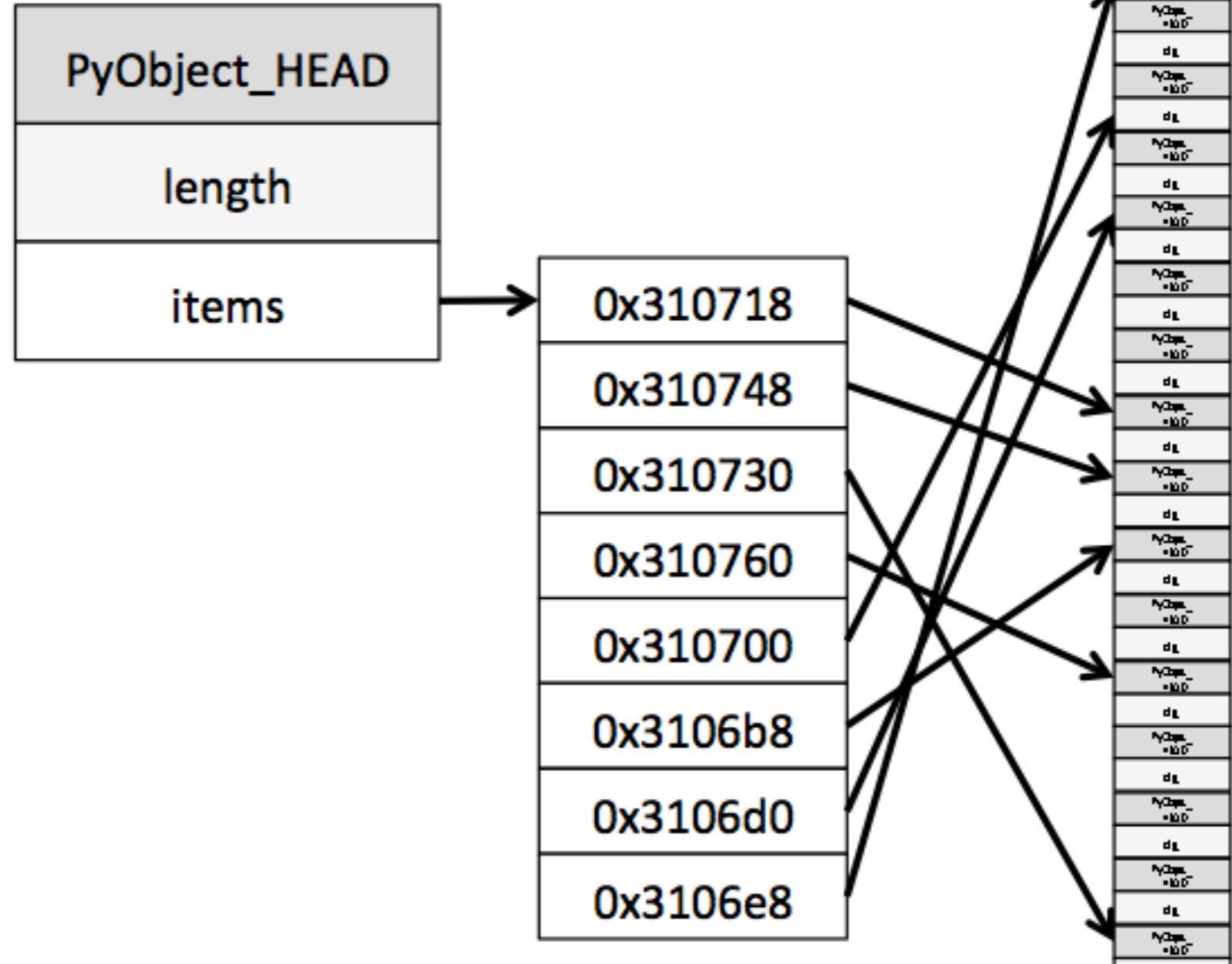


Numpy Array



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

Python List



중간 정리

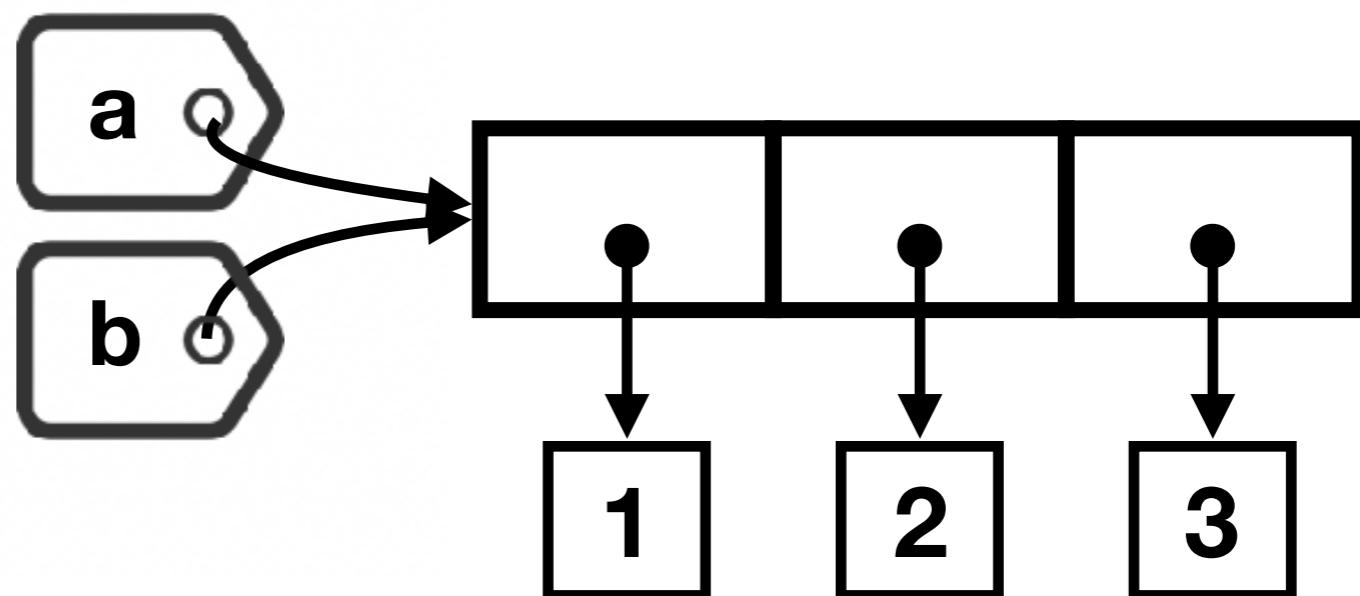
- 리바인딩은 시간이 많이 소요되더라
- 그래서 성능이 정말로 안 좋더라.
 - 파이썬은 느리다고 불평하는 경우가 이 경우이다
- 하지만 numpy를 사용하면 C와 비슷한 속도를 얻을 수 있다

라이브러리 최적화

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

파이썬 리스트

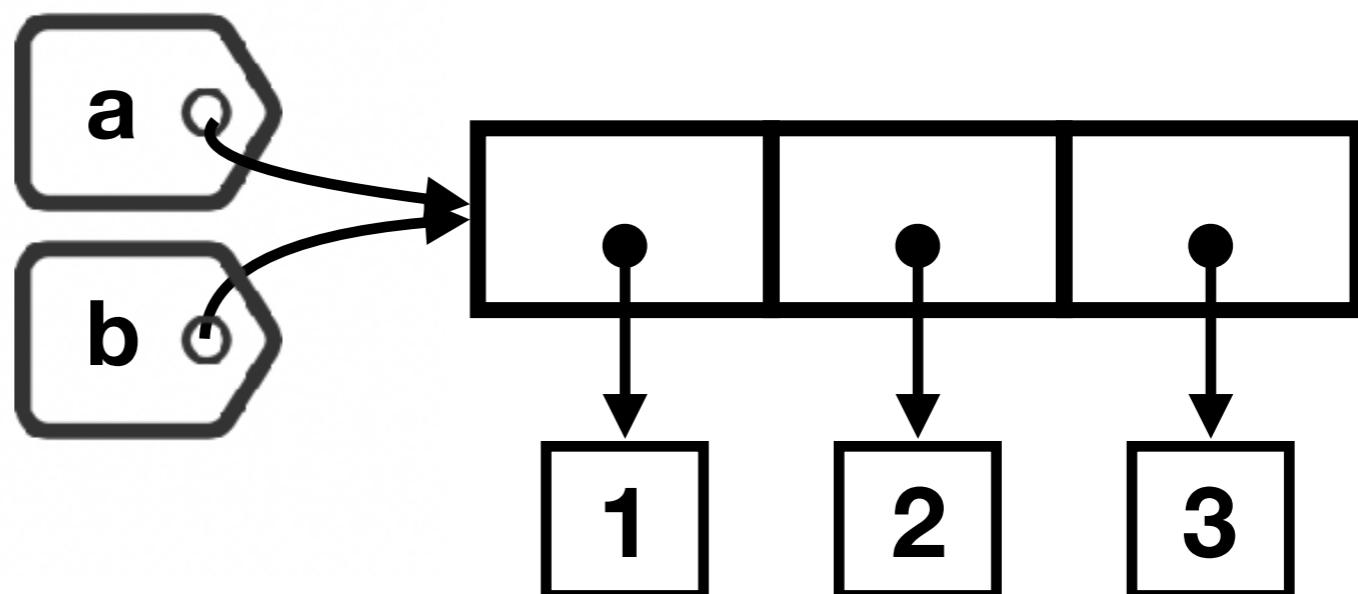
```
[>>> a = [1, 2, 3]
[>>> b = a
[>>> print(b)
[1, 2, 3]
[>>> id(a)
4512940232
[>>> id(b)
4512940232
```



파이썬 리스트

- 리스트는 가변(mutable) 자료형이다

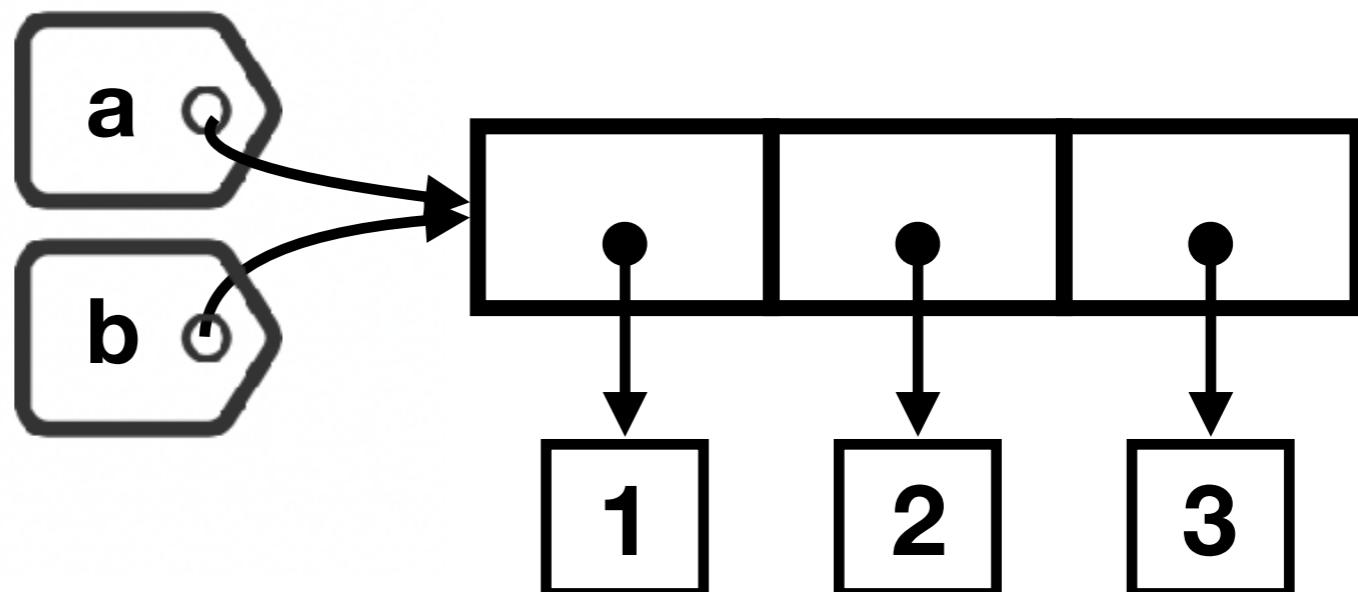
```
[>>> a = [1, 2, 3]
[>>> b = a
[>>> print(b)
[1, 2, 3]
[>>> id(a)
4512940232
[>>> id(b)
4512940232
```



파이썬 리스트

- 리스트는 가변(mutable) 자료형이다
- 리스트의 요소는 객체에 대한 참조값이다

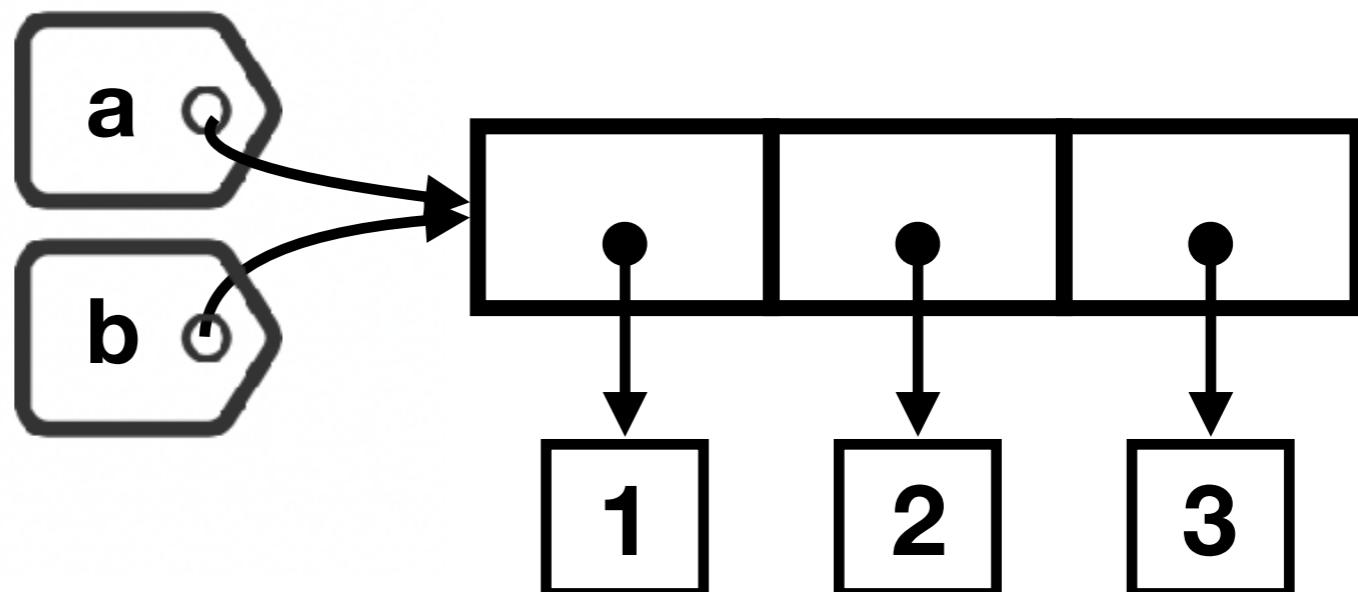
```
[>>> a = [1, 2, 3]
[>>> b = a
[>>> print(b)
[1, 2, 3]
[>>> id(a)
4512940232
[>>> id(b)
4512940232
```



파이썬 리스트

- 리스트는 가변(mutable) 자료형이다
- 리스트의 요소는 객체에 대한 참조값이다

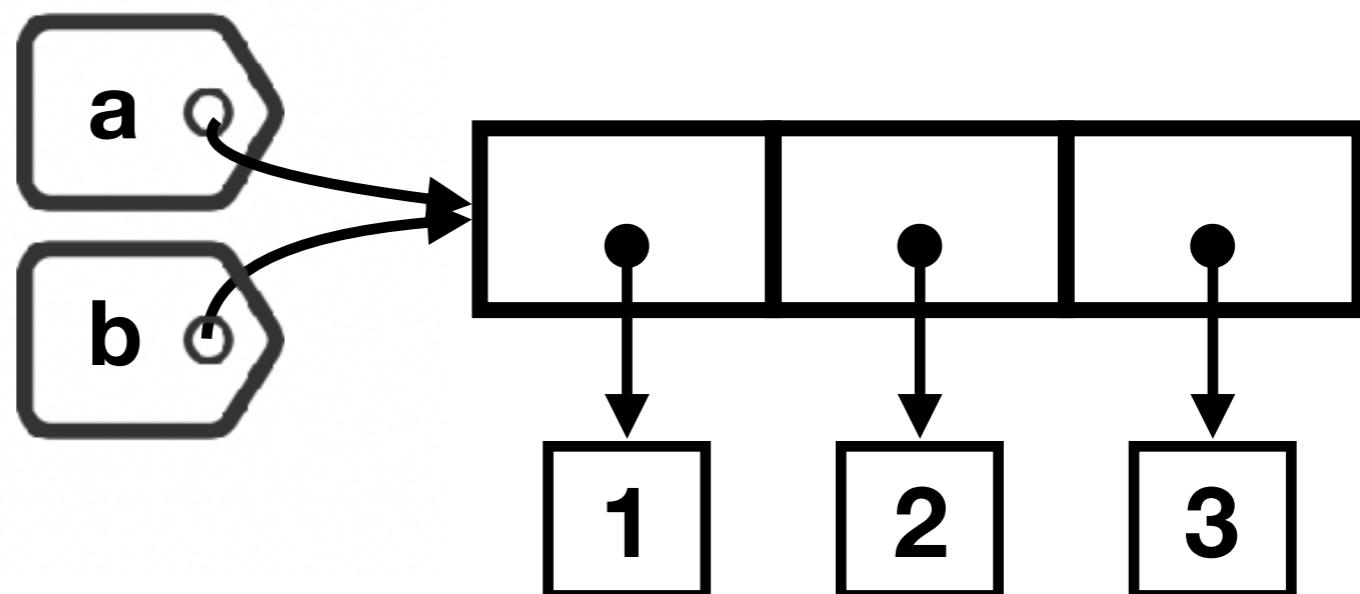
```
[>>> a = [1, 2, 3]
[>>> b = a
[>>> print(b)
[1, 2, 3]
[>>> id(a)
4512940232
[>>> id(b)
4512940232
```



파이썬 리스트

- 리스트는 가변(mutable) 자료형이다
- 리스트의 요소는 객체에 대한 참조값이다

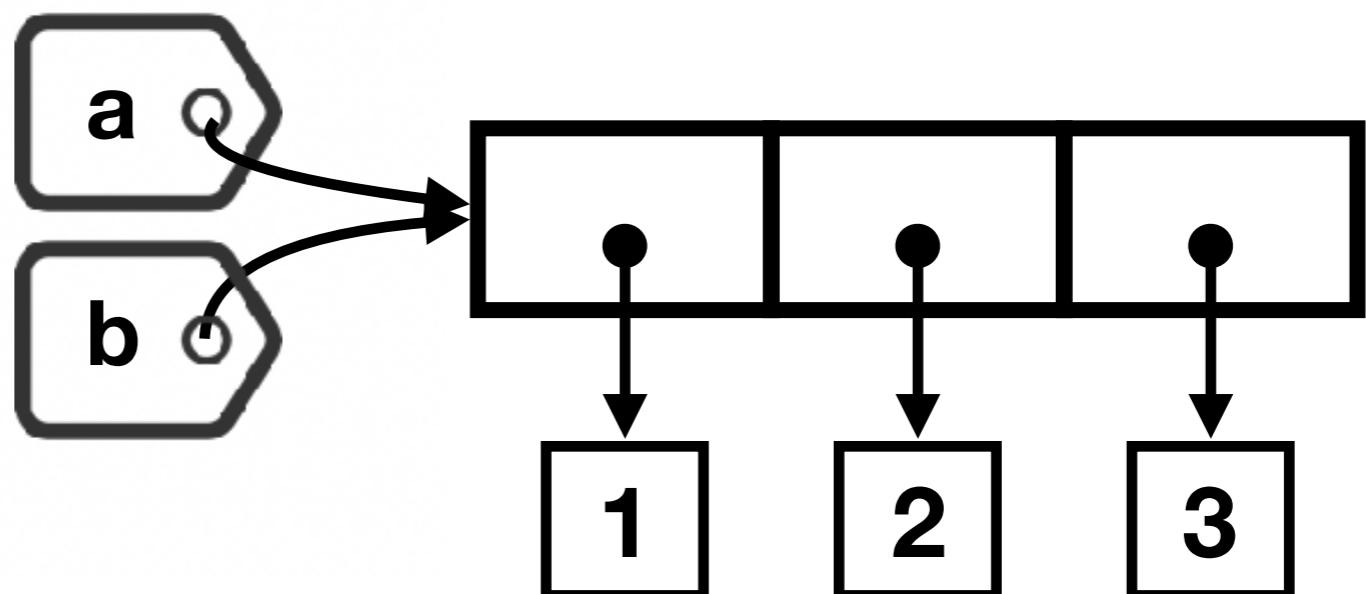
```
[>>> a = [1, 2, 3]
[>>> b = a
[>>> print(b)
[1, 2, 3]
[>>> id(a)
4512940232
[>>> id(b)
4512940232
```



파이썬 리스트

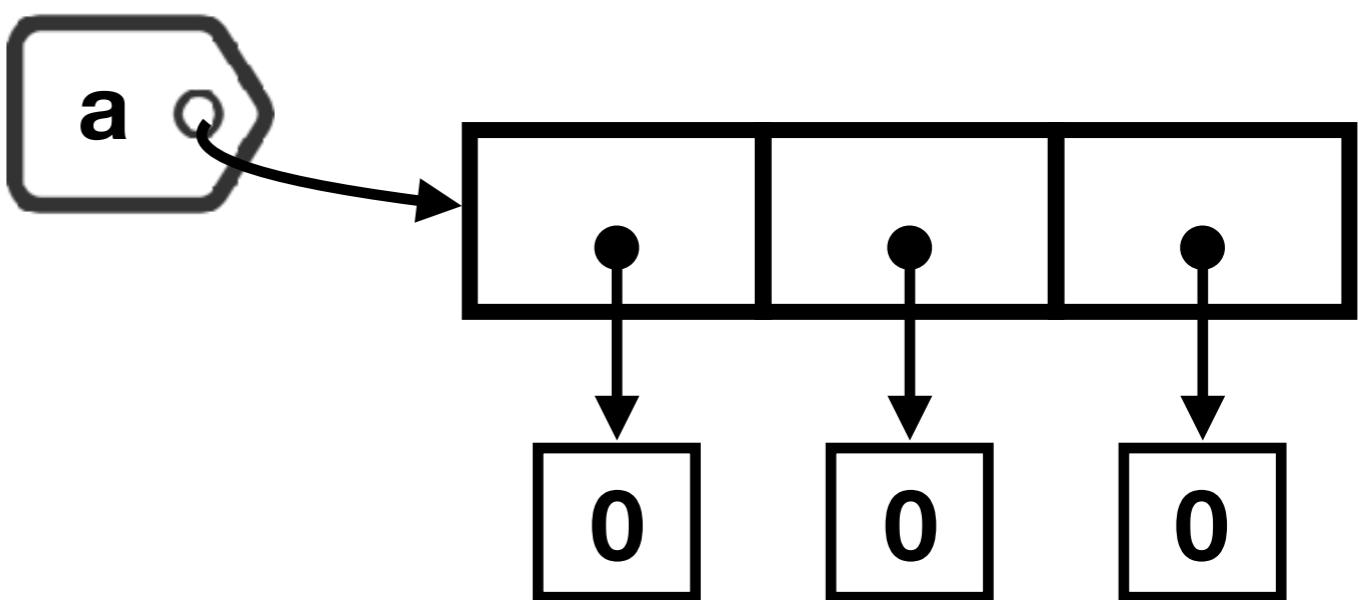
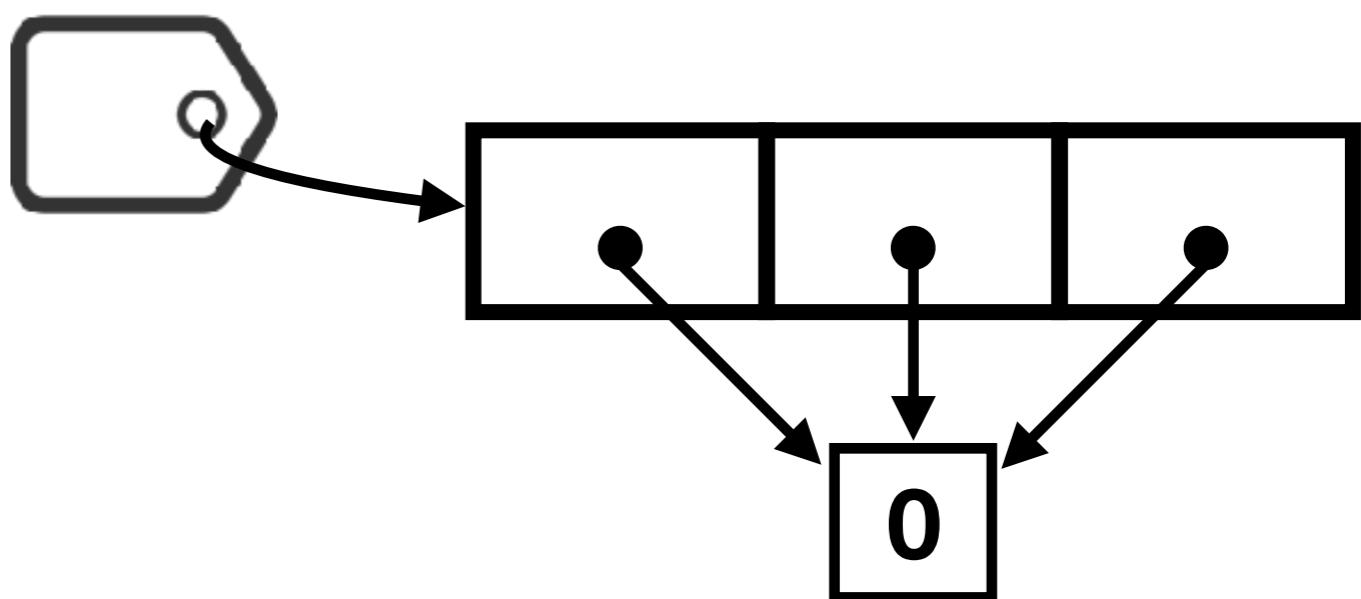
- 리스트는 가변(mutable) 자료형이다
- 리스트의 요소는 객체에 대한 참조값이다

```
[>>> a = [1, 2, 3]
[>>> b = a
[>>> print(b)
[1, 2, 3]
[>>> id(a)
4512940232
[>>> id(b)
4512940232
```



Quiz

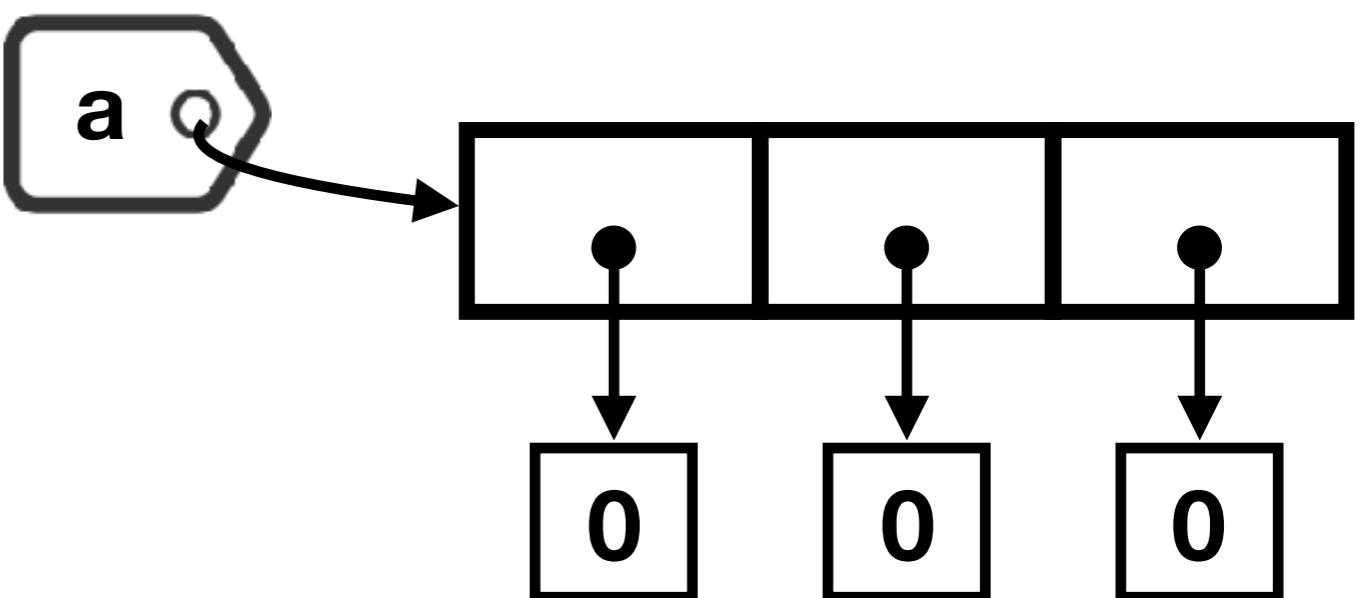
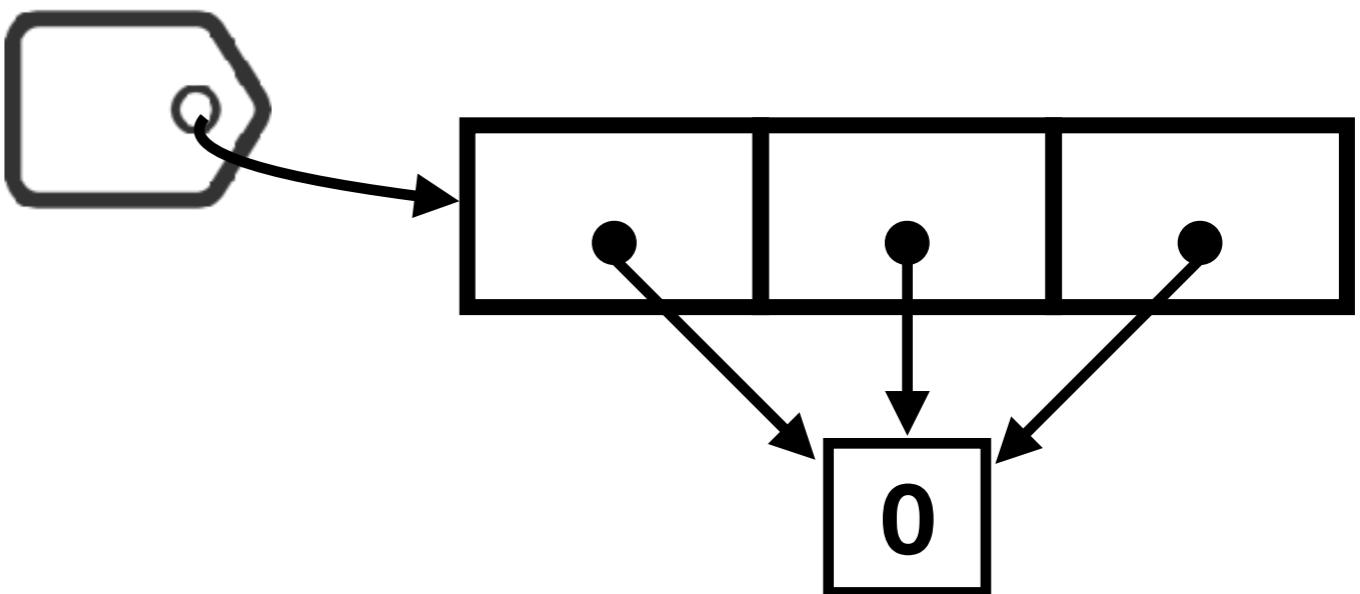
```
>>> a = [0, 0, 0]  
>>>
```



Quiz

```
>>> a = [0, 0, 0]  
>>>
```

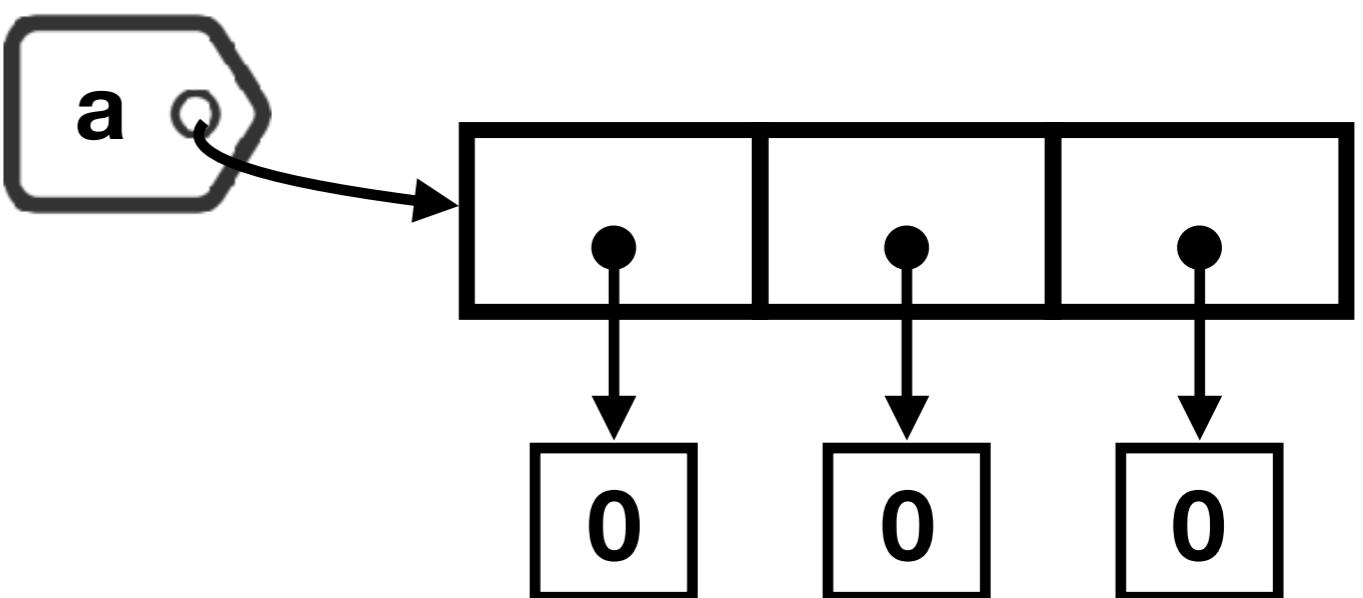
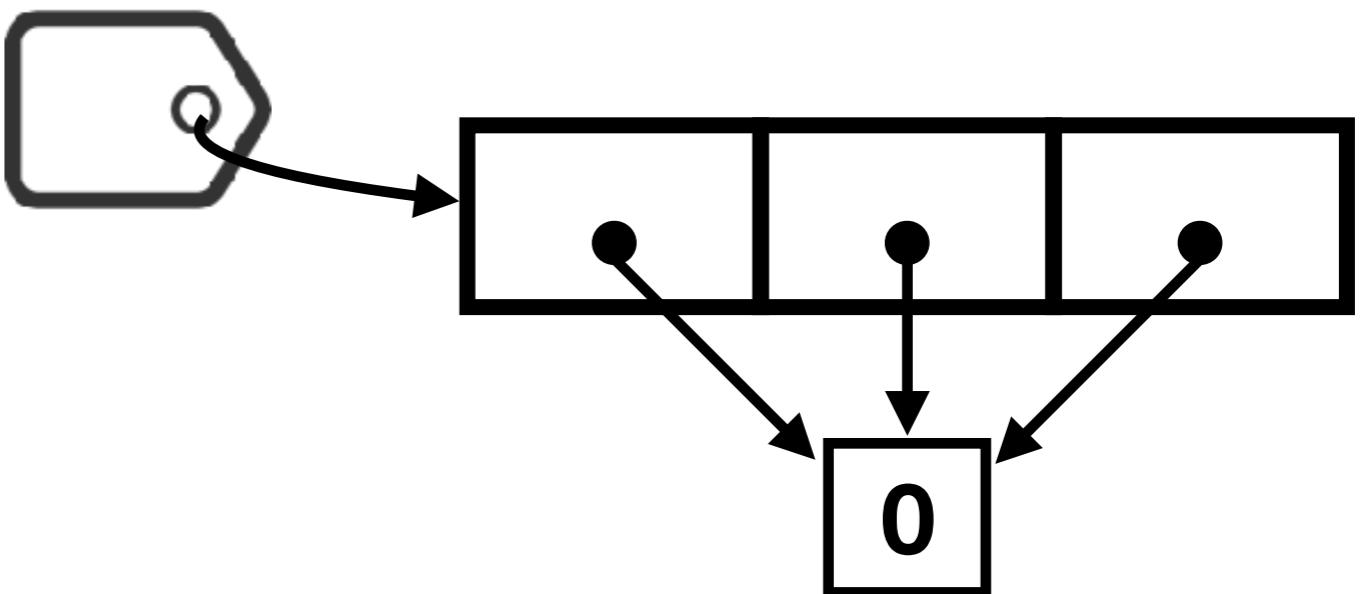
의 결과는?



Quiz

```
>>> a = [0, 0, 0]  
>>>
```

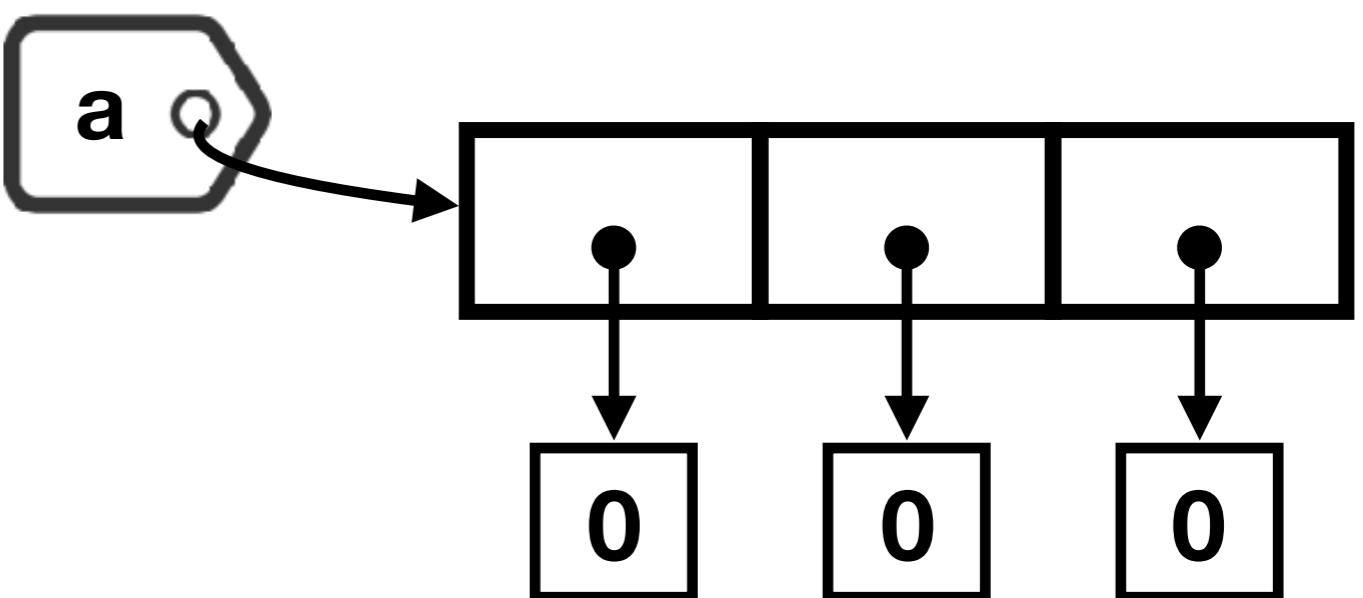
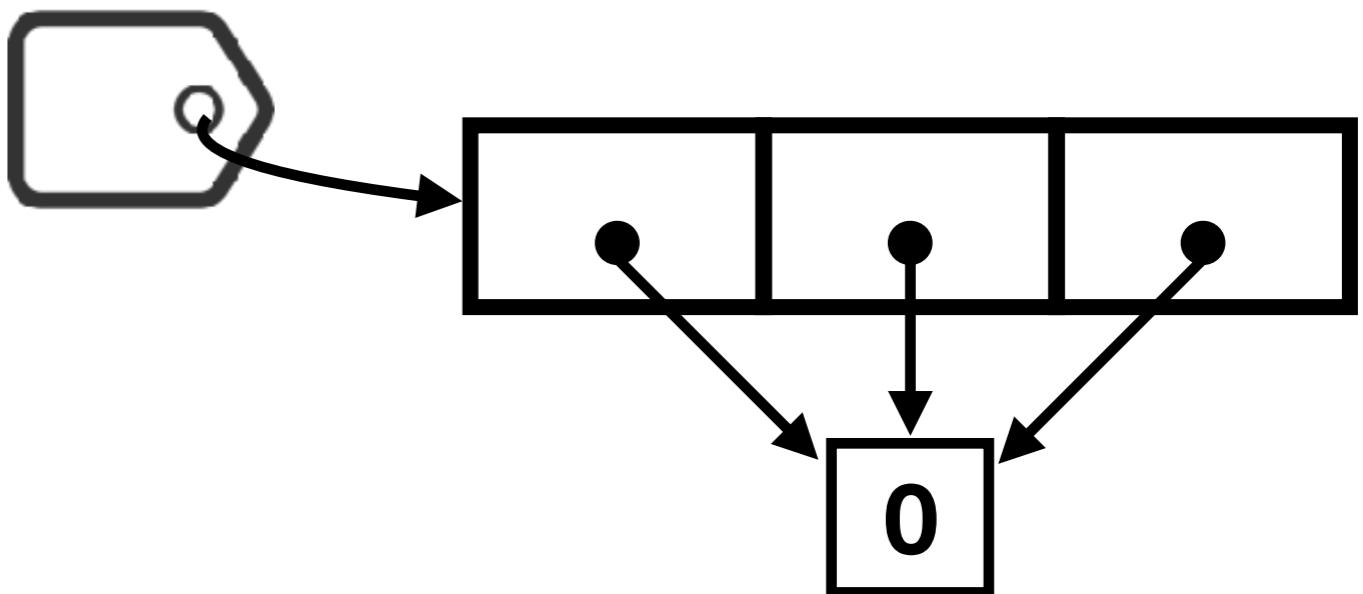
의 결과는?



Quiz

```
>>> a = [0, 0, 0]  
>>>
```

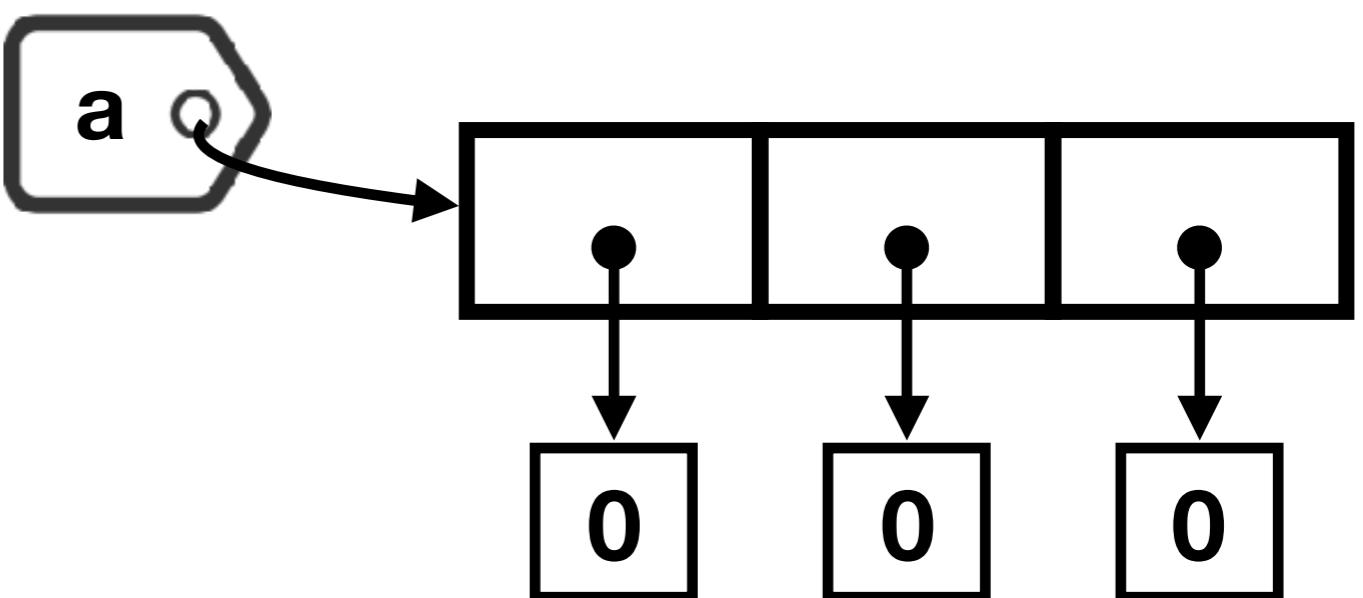
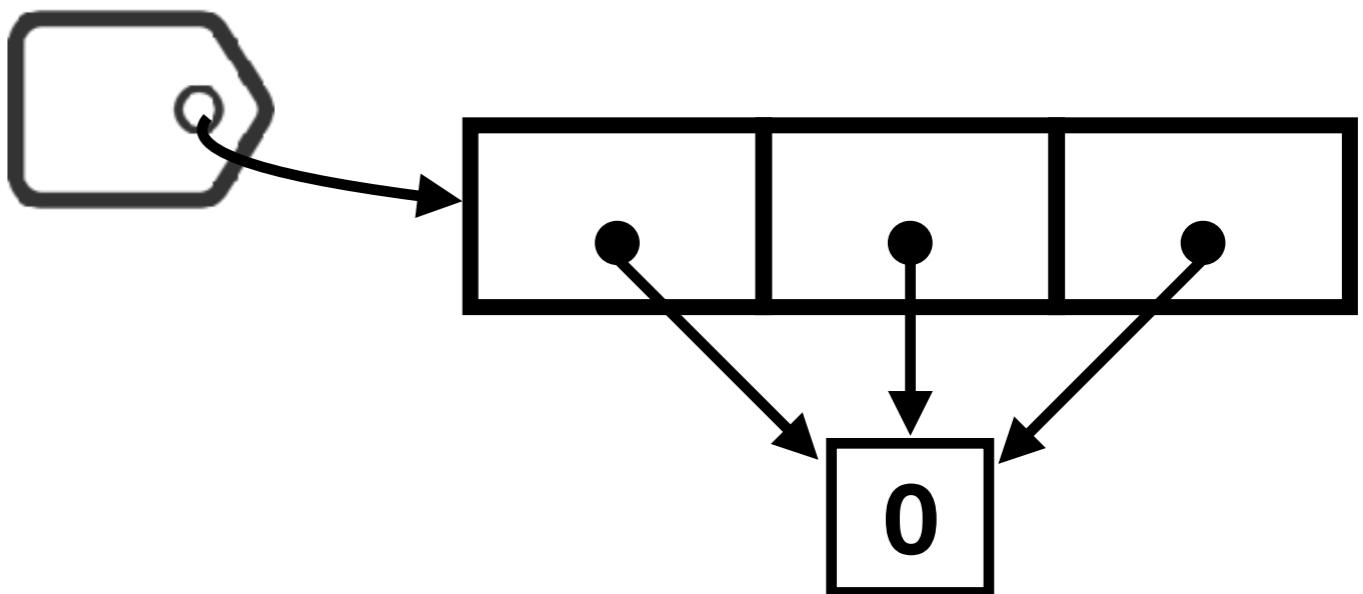
의 결과는?



Quiz

```
>>> a = [0, 0, 0]  
>>>
```

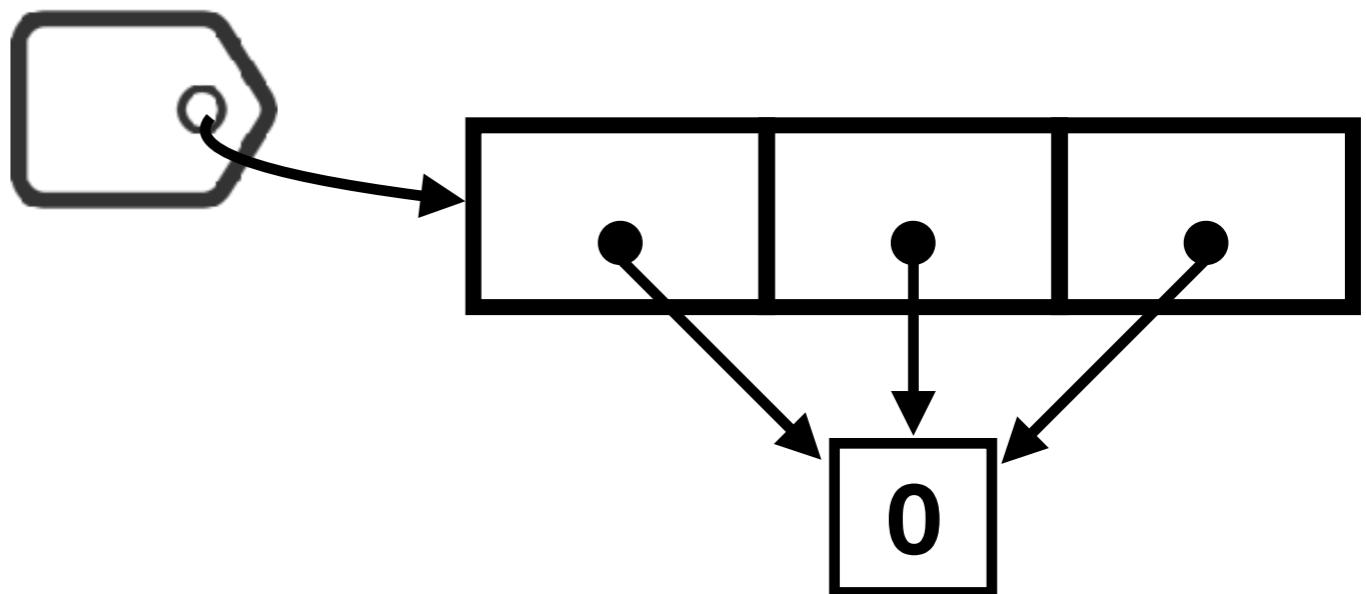
의 결과는?



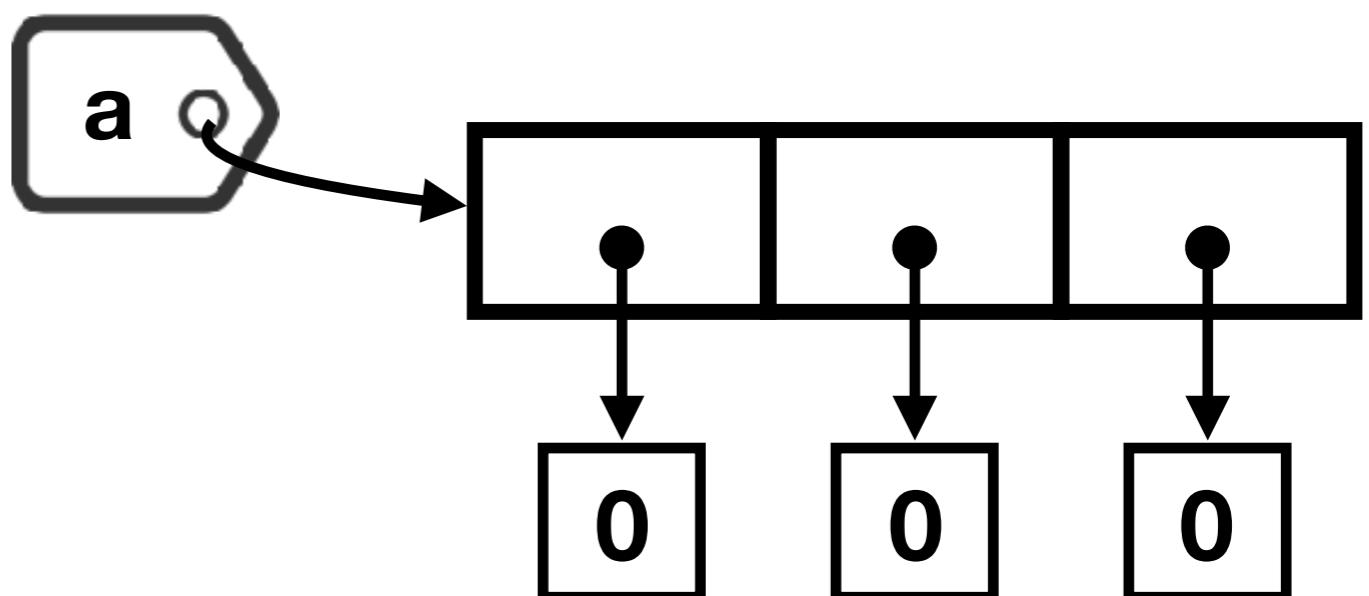
Quiz

```
>>> a = [0, 0, 0]  
>>>
```

의 결과는?



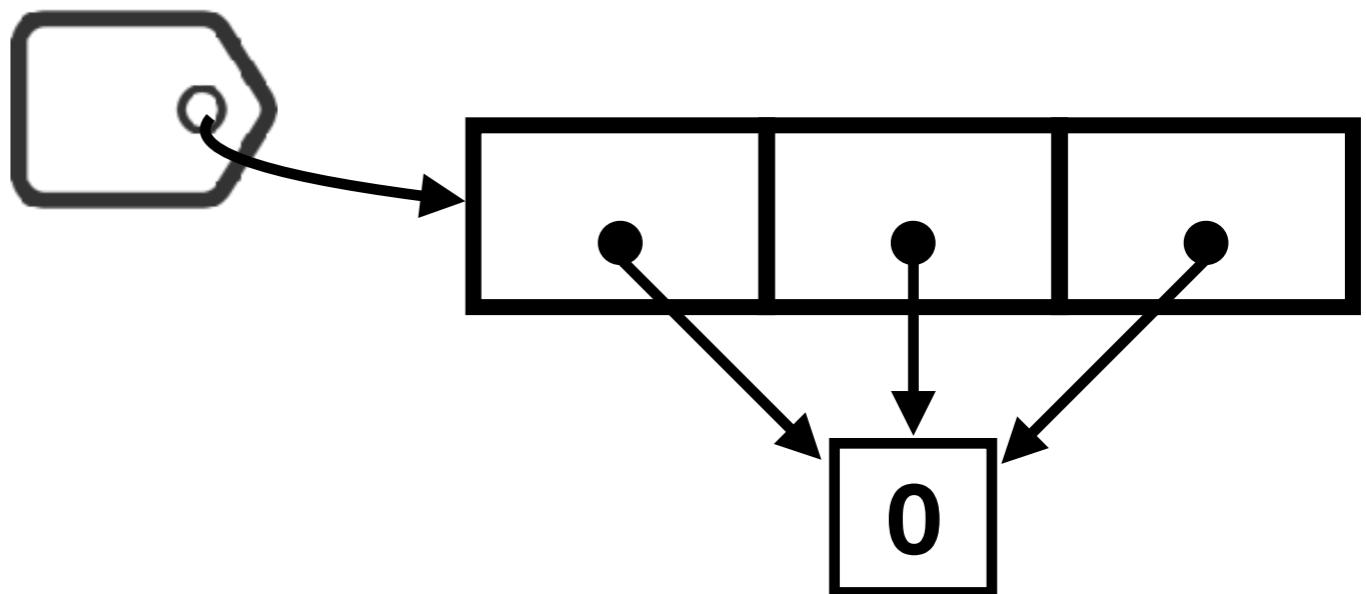
1) 하나의 객체를 리스트 요소가 참조한다



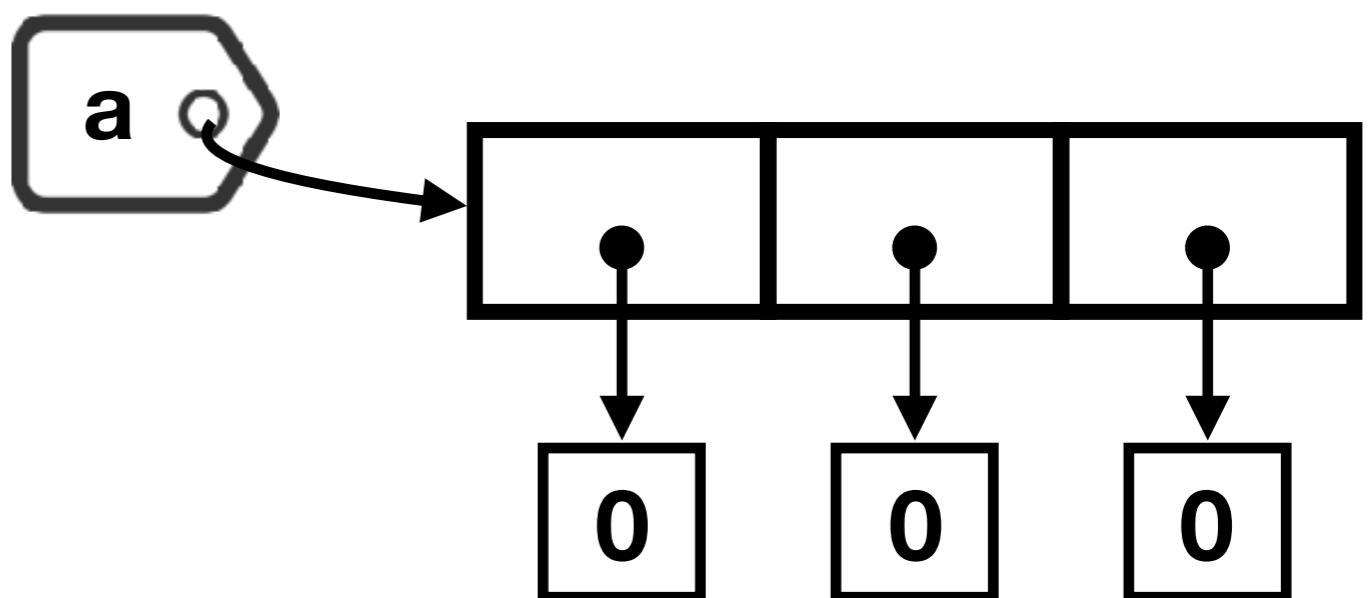
Quiz

```
>>> a = [0, 0, 0]  
>>>
```

의 결과는?



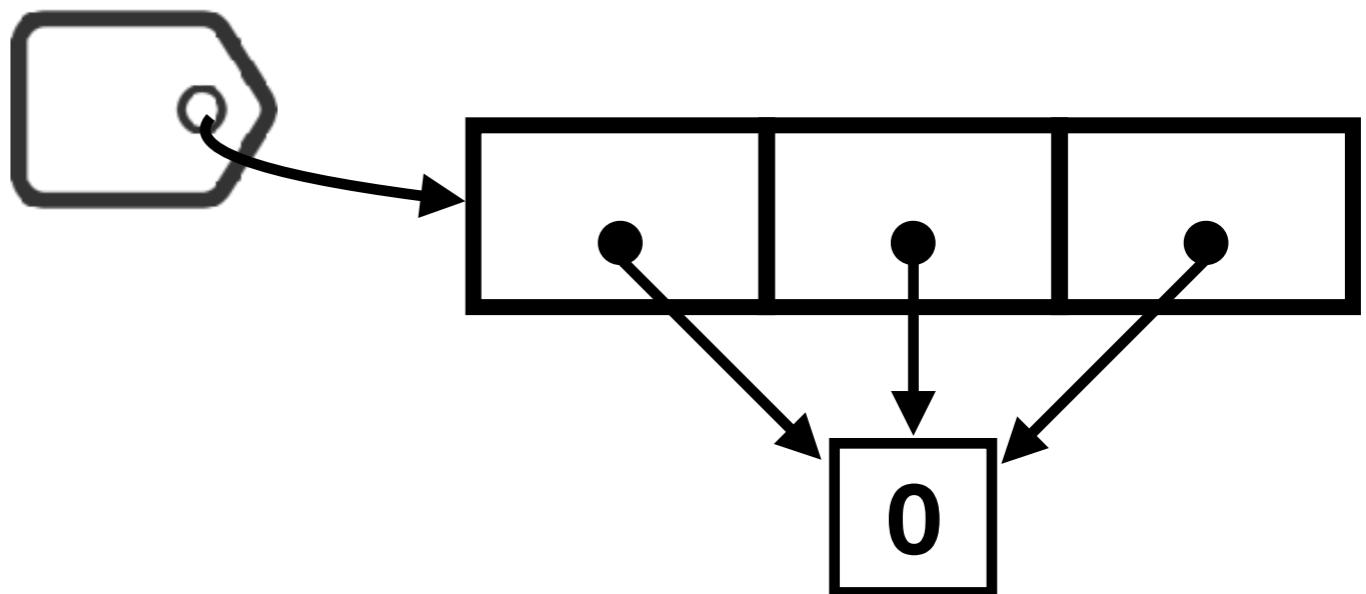
1) 하나의 객체를 리스트 요소가 참조한다



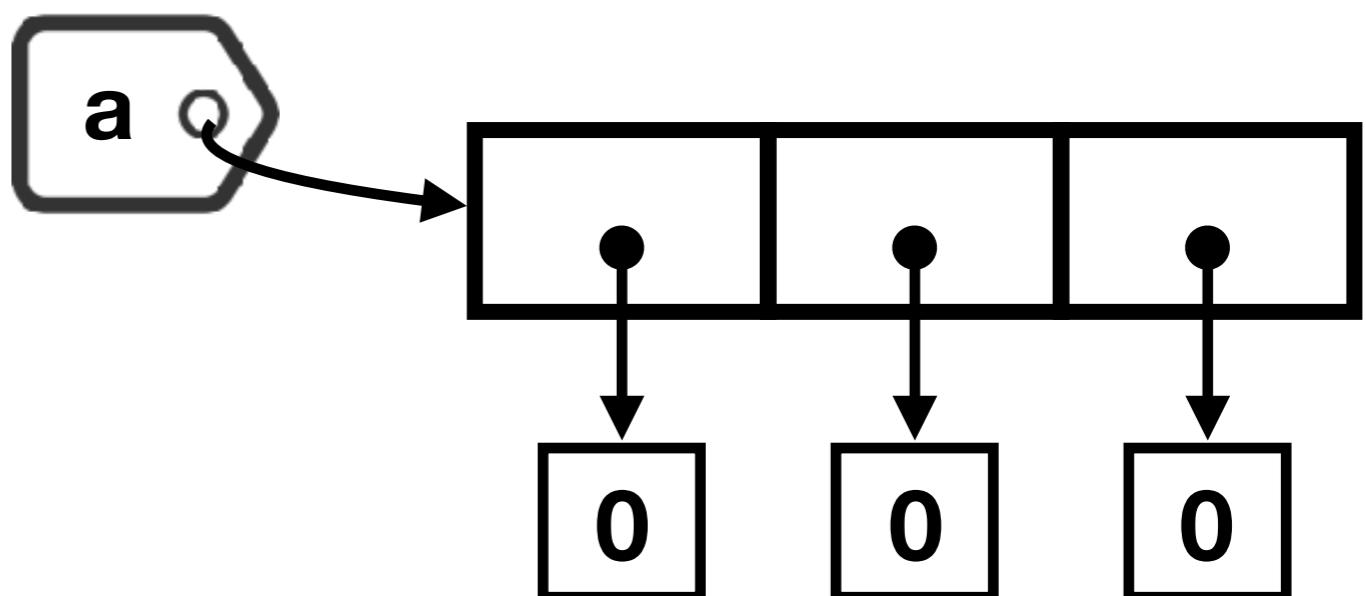
Quiz

```
>>> a = [0, 0, 0]  
>>>
```

의 결과는?



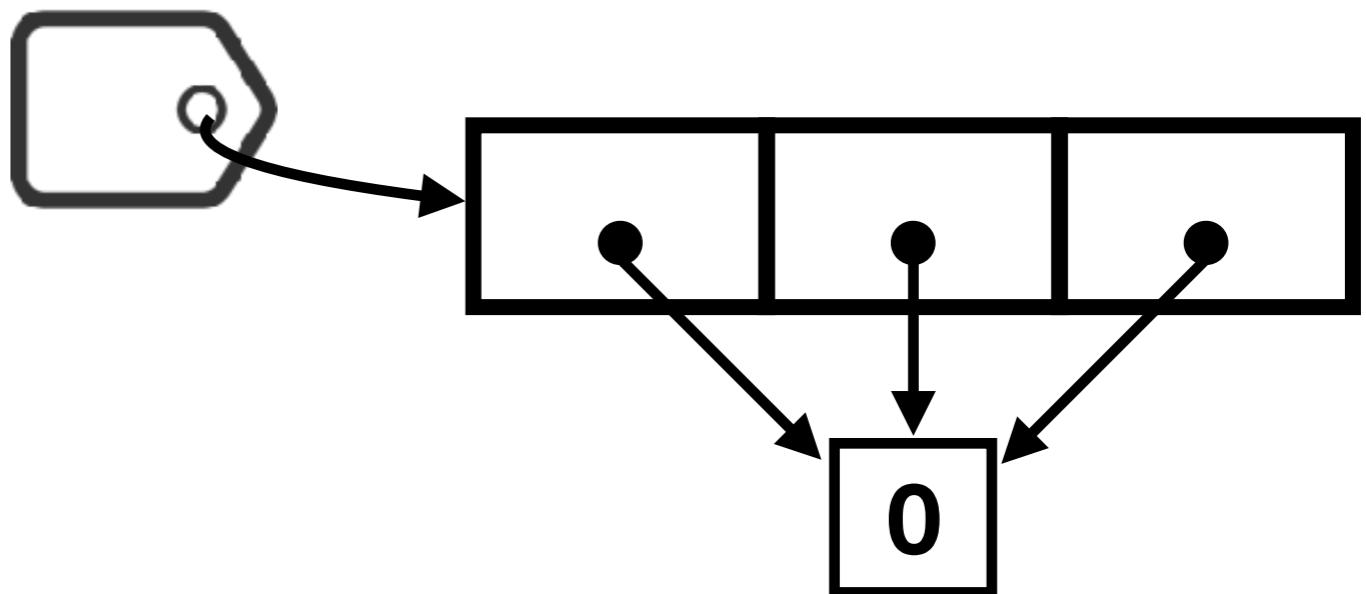
1) 하나의 객체를 리스트 요소가 참조한다



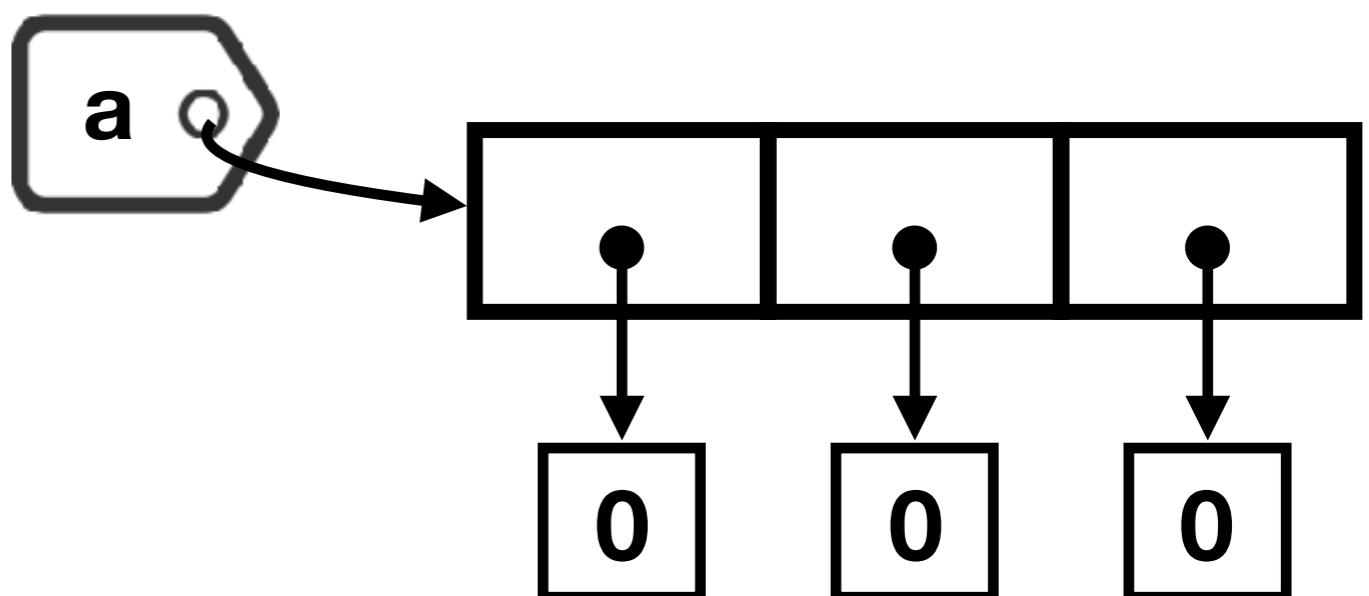
Quiz

```
>>> a = [0, 0, 0]  
>>>
```

의 결과는?



1) 하나의 객체를 리스트 요소가 참조한다

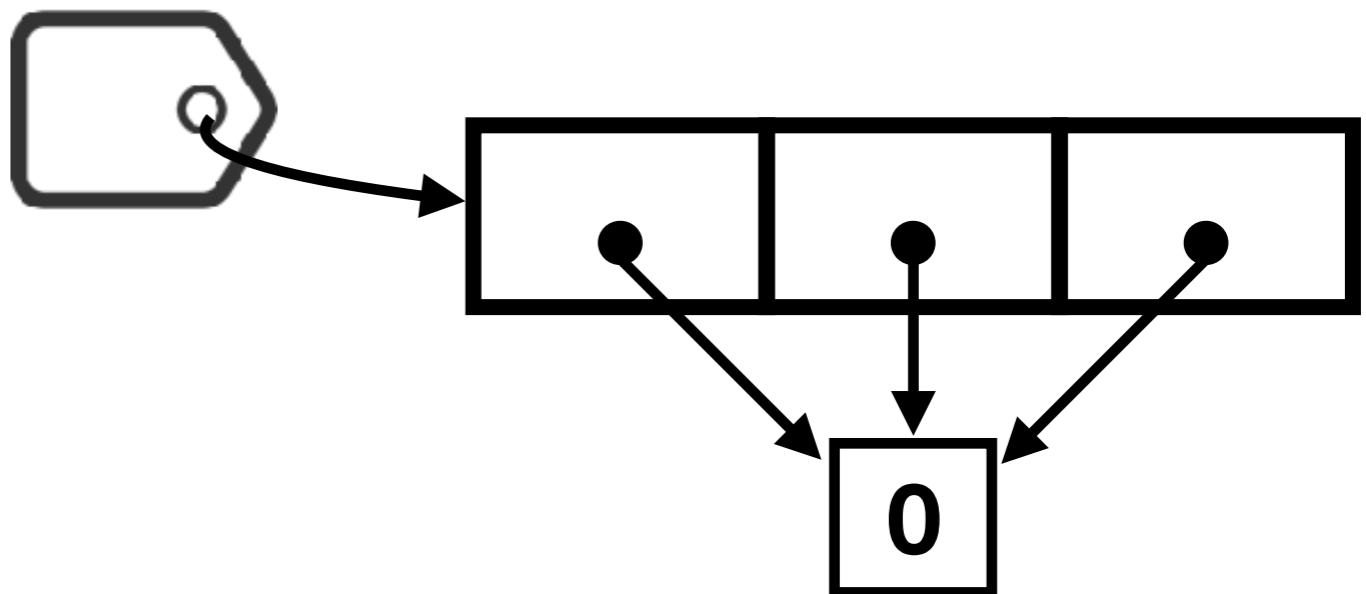


Quiz

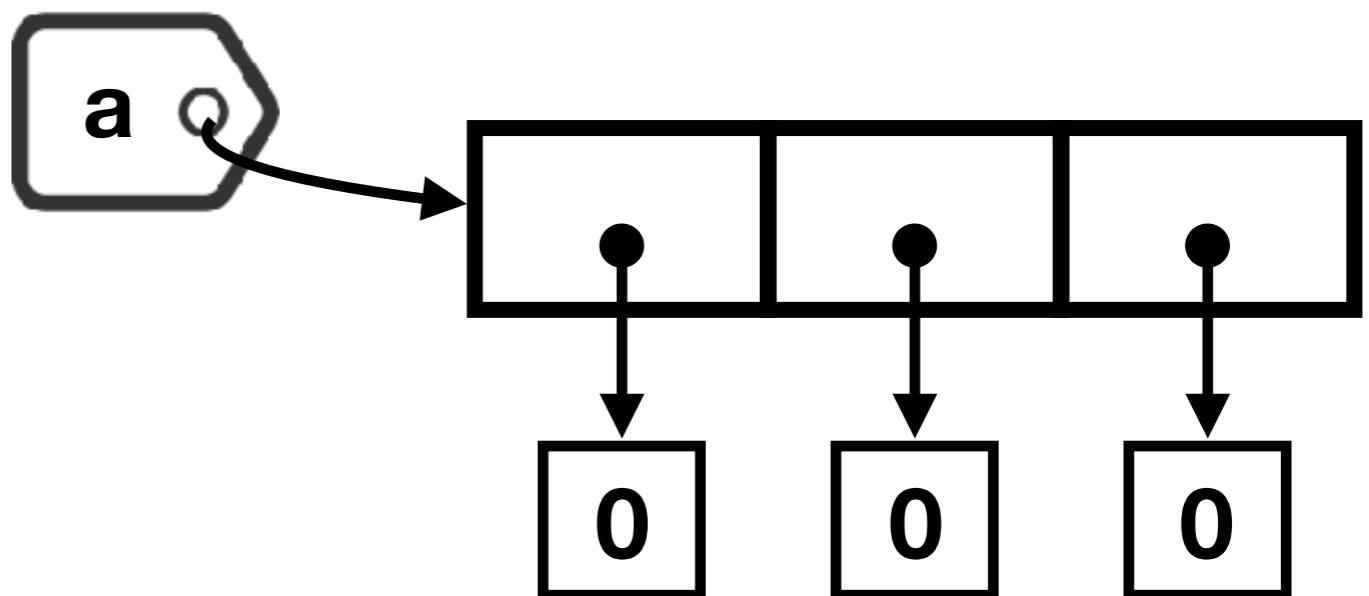
```
>>> a = [0, 0, 0]
```

```
>>>
```

의 결과는?



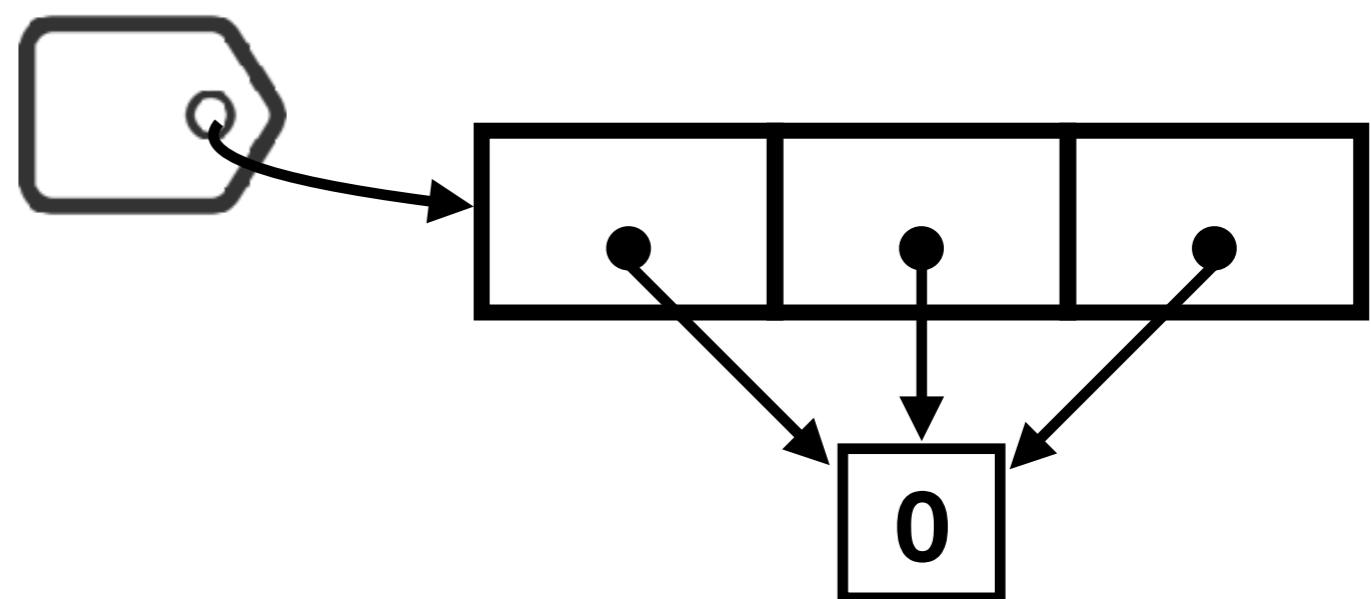
1) 하나의 객체를 리스트 요소가 참조한다



2) 3개의 객체를 리스트 요소들이 각각 참조한다

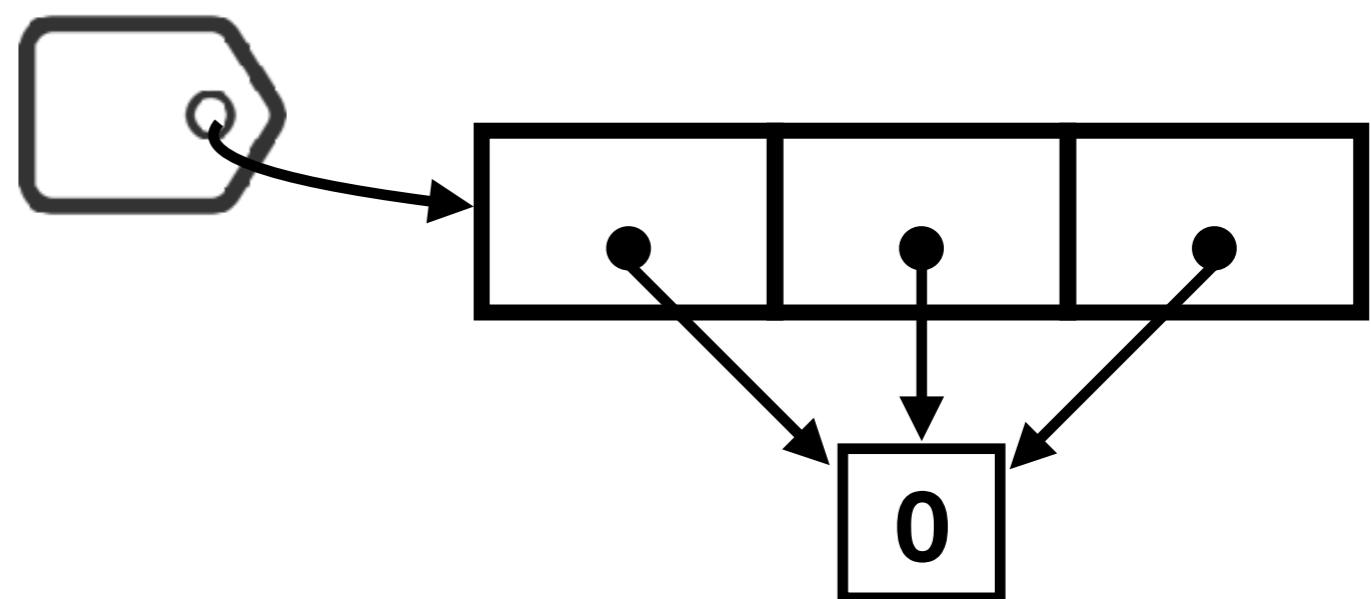
정답

```
>>> a = [0, 0, 0]  
>>>
```



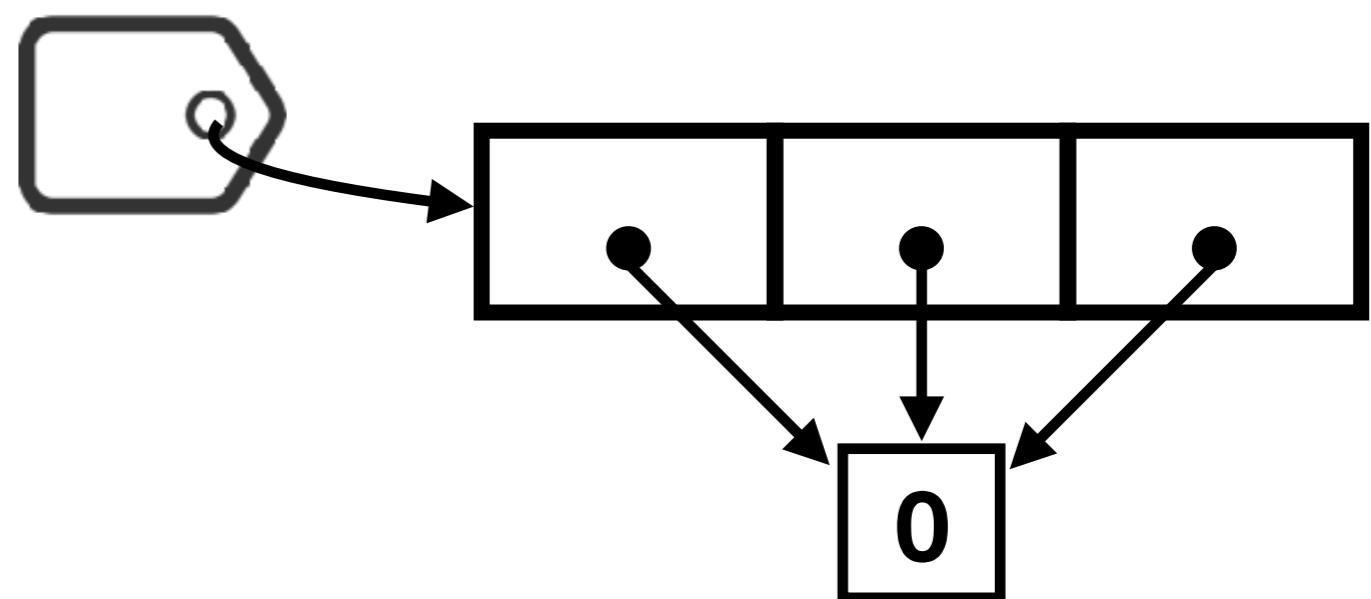
정답

```
>>> a = [0, 0, 0]  
>>>
```



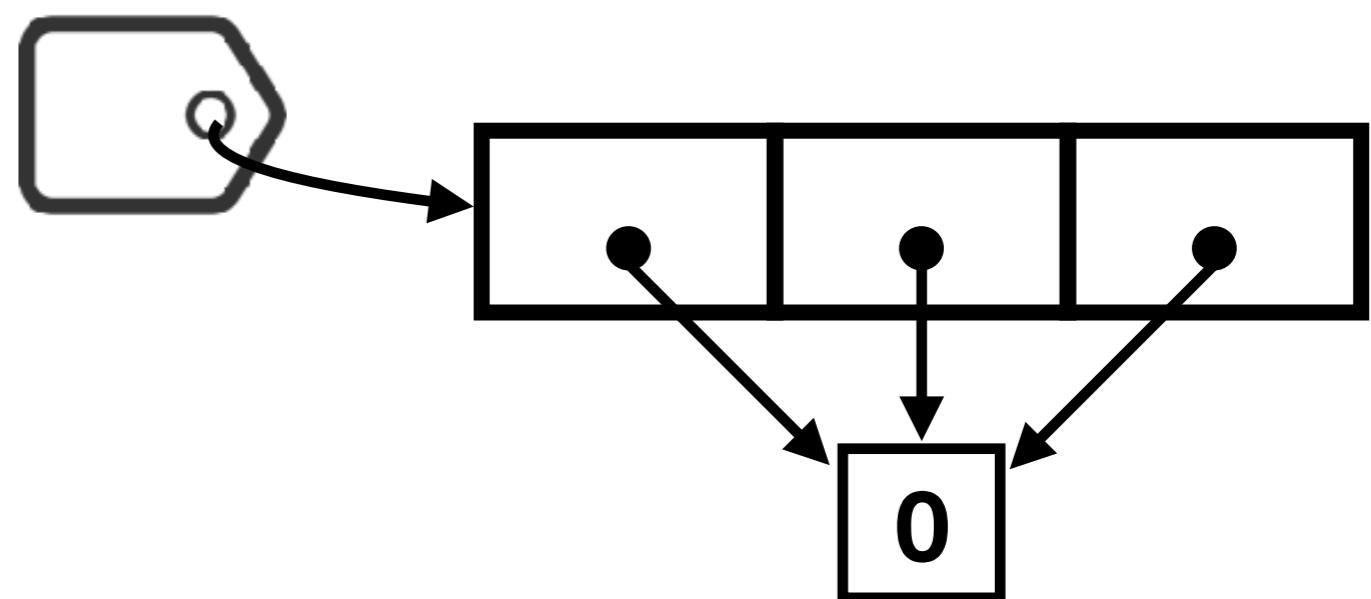
정답

```
>>> a = [0, 0, 0]  
>>>
```



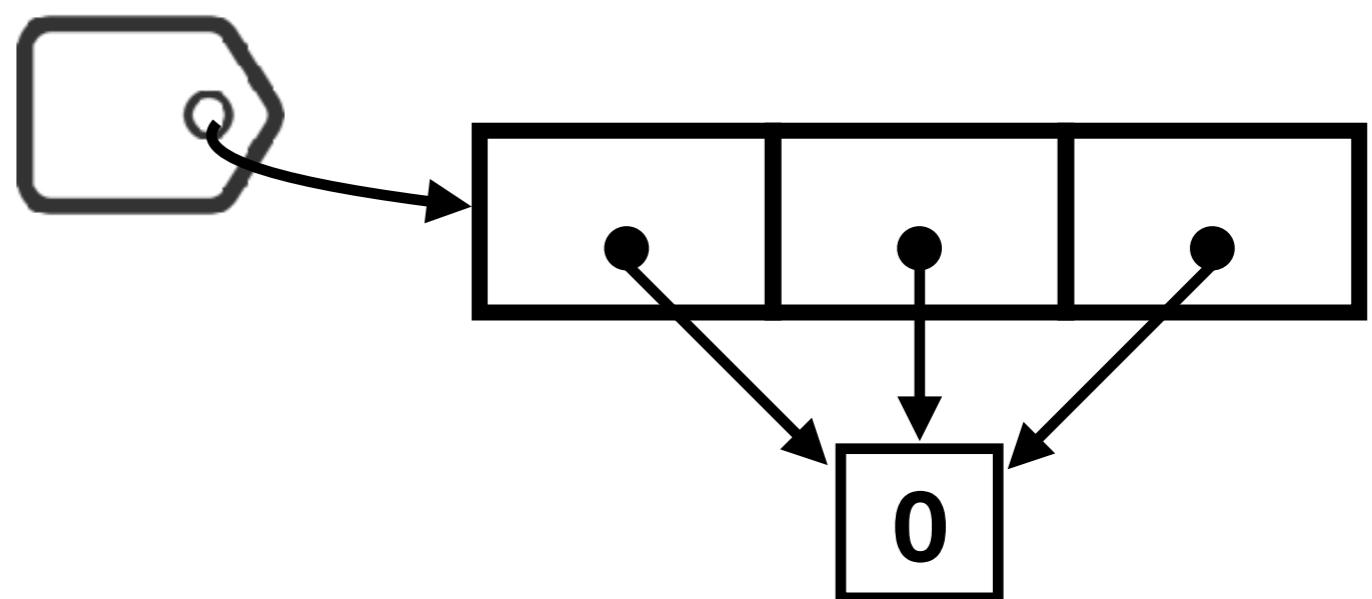
정답

```
>>> a = [0, 0, 0]  
>>>
```



정답

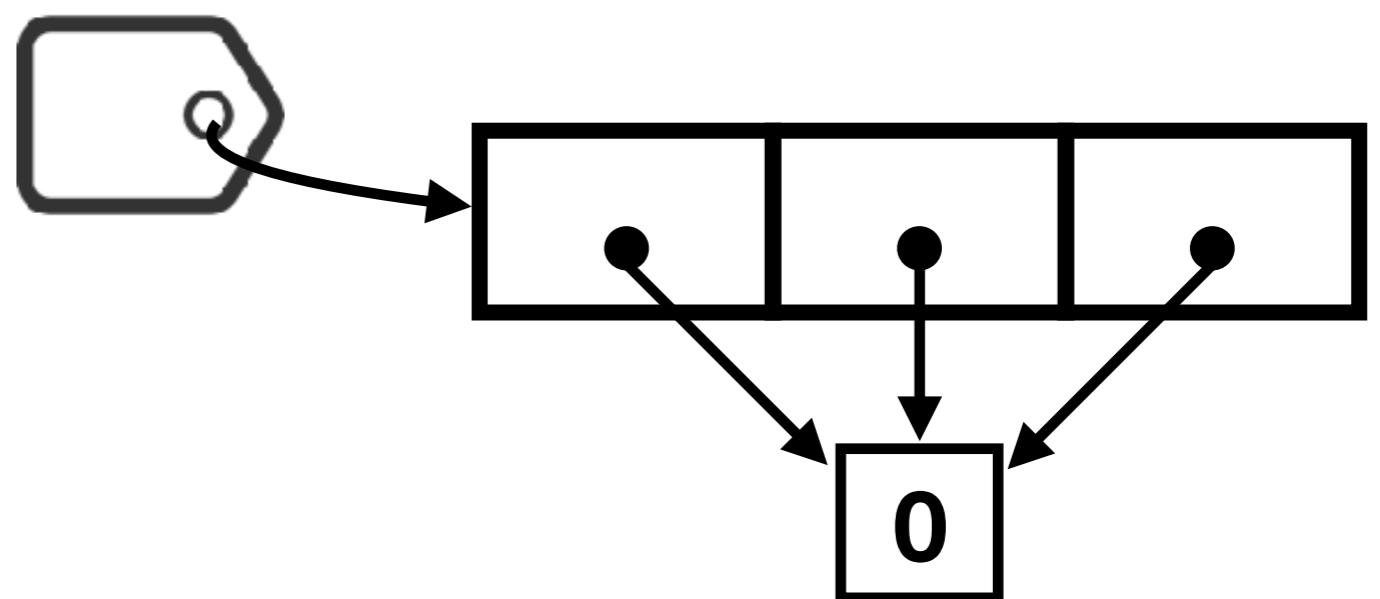
```
>>> a = [0, 0, 0]  
>>>
```



1) 하나의 객체를 리스트 요소가 참조한다

정답

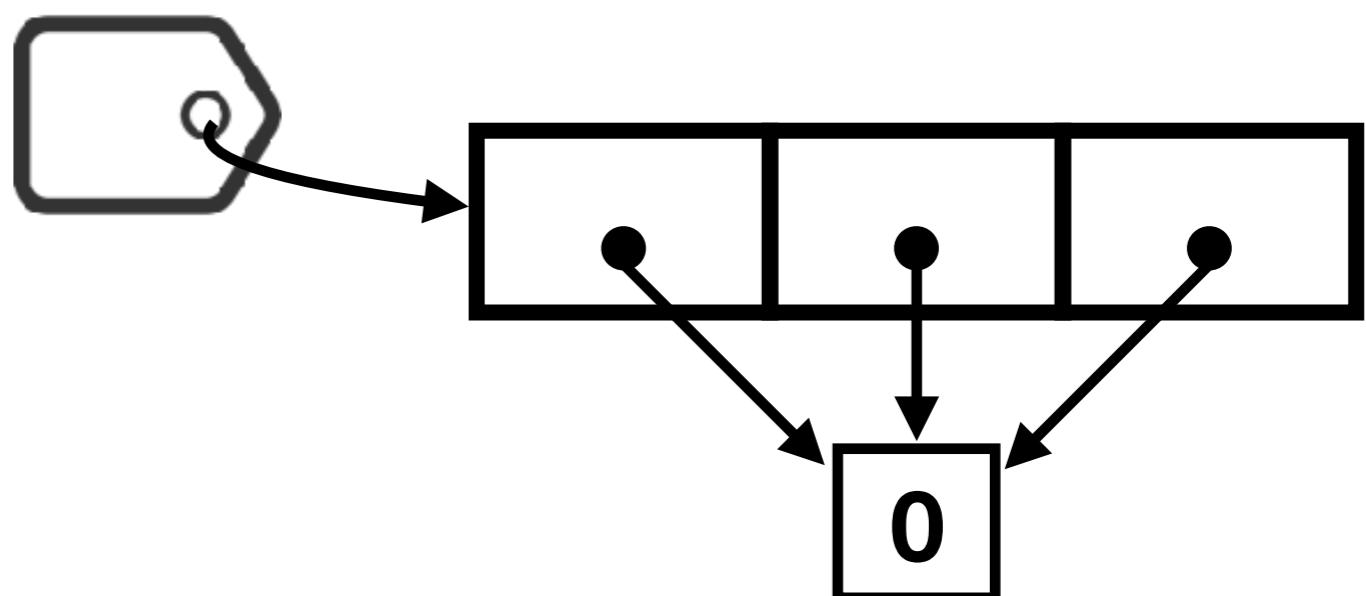
```
>>> a = [0, 0, 0]
>>>
>>> id(a[0])
4402199312
>>> id(a[1])
4402199312
>>> id(a[2])
4402199312
```



1) 하나의 객체를 리스트 요소가 참조한다

정답

```
>>> a = [0, 0, 0]
>>>
>>> id(a[0])
4402199312
>>> id(a[1])
4402199312
>>> id(a[2])
4402199312
```



1) 하나의 객체를 리스트 요소가 참조한다

다차원 리스트

1 : board = [[0] * cols] * rows

2 : board = [[0] * cols for _ in range(rows)]

둘다 2차원 리스트인데???

두 가지 방법을 비교해보자!! ◉

어떤 차이점이 있을까?

어느쪽이 더 빠를까?

```
1 import time, sys  
2  
3 rows, cols = 300, 3000  
4 start = time.time()  
5 board = [[0] * cols] * rows  
6 end = time.time()  
7 print("수행시간 : {:.10f}".format(end - start)).
```

수행시간 : 0.000068



```
1 import time, sys  
2  
3 rows, cols = 300, 3000  
4 start = time.time()  
5 board = [[0] * cols for _ in range(rows)]  
6 end = time.time()  
7 print("수행시간 : {:.10f}".format(end - start)).
```

수행시간 : 0.007204

어느쪽이 더 빠를까?

```
1 import time, sys
2
3 rows, cols = 300, 3000
4 start = time.time()
5 board = [[0] * cols] * rows
6 end = time.time()
7 print("수행시간 : {:.10f}".format(end - start)).
```

수행시간 : 0.000068



```
1 import time, sys
2
3 rows, cols = 300, 3000
4 start = time.time()
5 board = [[0] * cols for _ in range(rows)]
6 end = time.time()
7 print("수행시간 : {:.10f}".format(end - start)).
```

수행시간 : 0.007204

어느쪽이 더 빠를까?

```
1 import time, sys
2
3 rows, cols = 300, 3000
4 start = time.time()
5 board = [[0] * cols] * rows
6 end = time.time()
7 print("수행시간 : {:.10f}".format(end - start)).
```

수행시간 : 0.000068



```
1 import time, sys
2
3 rows, cols = 300, 3000
4 start = time.time()
5 board = [[0] * cols for _ in range(rows)]
6 end = time.time()
7 print("수행시간 : {:.10f}".format(end - start)).
```

수행시간 : 0.007204

어느쪽이 더 빠를까?

```
1 import time, sys
2
3 rows, cols = 300, 3000
4 start = time.time()
5 board = [[0] * cols] * rows
6 end = time.time()
7 print("수행시간 : {:.10f}".format(end - start)).
```

수행시간 : 0.000068



```
1 import time, sys
2
3 rows, cols = 300, 3000
4 start = time.time()
5 board = [[0] * cols for _ in range(rows)]
6 end = time.time()
7 print("수행시간 : {:.10f}".format(end - start)).
```

수행시간 : 0.007204

어느쪽이 더 빠를까?

```
1 import time, sys
2
3 rows, cols = 300, 3000
4 start = time.time()
5 board = [[0] * cols] * rows
6 end = time.time()
7 print("수행시간 : {:.10f}".format(end - start)).
```

수행시간 : 0.000068



```
1 import time, sys
2
3 rows, cols = 300, 3000
4 start = time.time()
5 board = [[0] * cols for _ in range(rows)]
6 end = time.time()
7 print("수행시간 : {:.10f}".format(end - start)).
```

수행시간 : 0.007204

어느쪽이 더 빠를까?

```
1 import time, sys
2
3 rows, cols = 300, 3000
4 start = time.time()
5 board = [[0] * cols] * rows
6 end = time.time()
7 print("수행시간 : {:.10f}".format(end - start)).
```

수행시간 : 0.000068



```
1 import time, sys
2
3 rows, cols = 300, 3000
4 start = time.time()
5 board = [[0] * cols for _ in range(rows)]
6 end = time.time()
7 print("수행시간 : {:.10f}".format(end - start)).
```

수행시간 : 0.007204

1번이 더 빠른 이유는

1 : board = [[0] * cols] * rows

2 : board = [[0] * cols for _ in range(rows)]

rows, cols = 3, 3인 경우에 대해 살펴보자

어떤 차이점?

```
board = [[0] * 3] * 3
```

어떤 차이점?

board = [[0] * 3] * 3

[0] * 3

어떤 차이점?

board = [[0] * 3] * 3

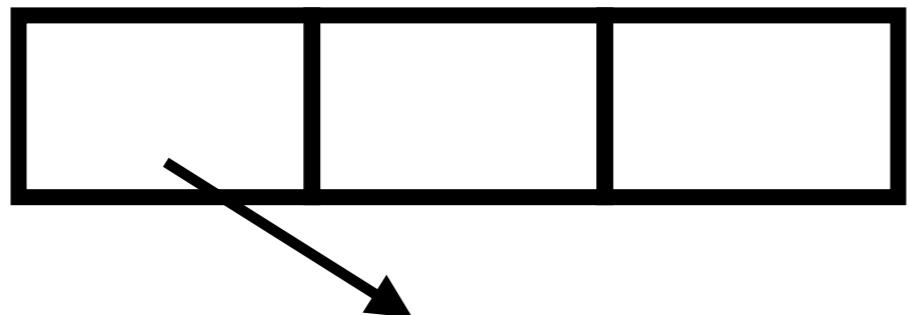
[0] * 3



어떤 차이점?

board = [[0] * 3] * 3

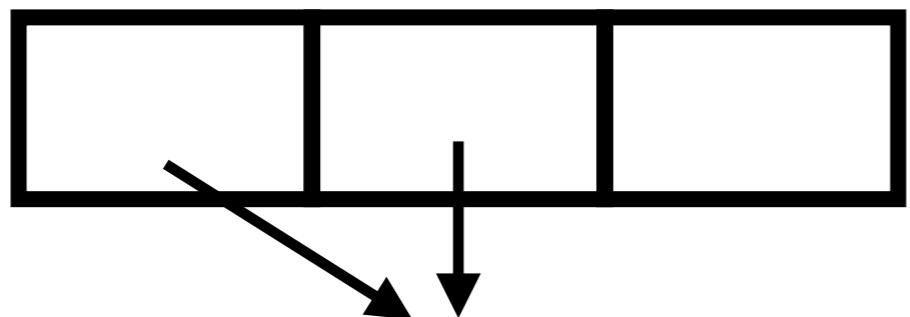
[0] * 3



어떤 차이점?

board = [[0] * 3] * 3

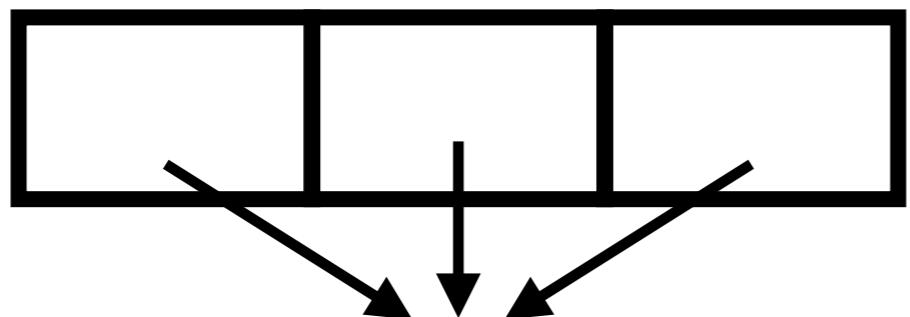
[0] * 3



어떤 차이점?

board = [[0] * 3] * 3

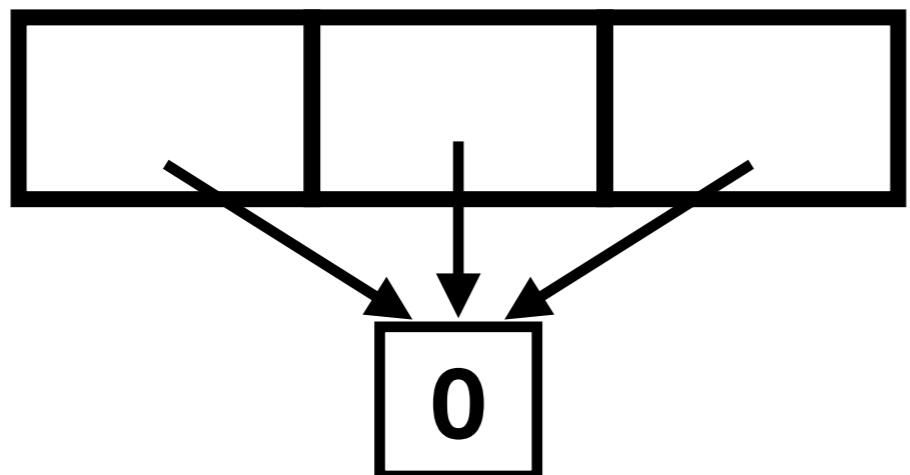
[0] * 3



어떤 차이점?

board = [[0] * 3] * 3

[0] * 3

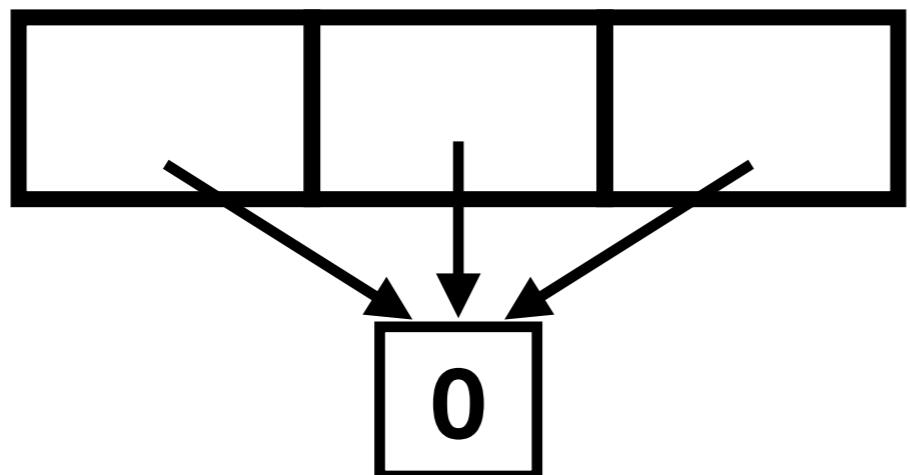


어떤 차이점?

board = [[0] * 3] * 3

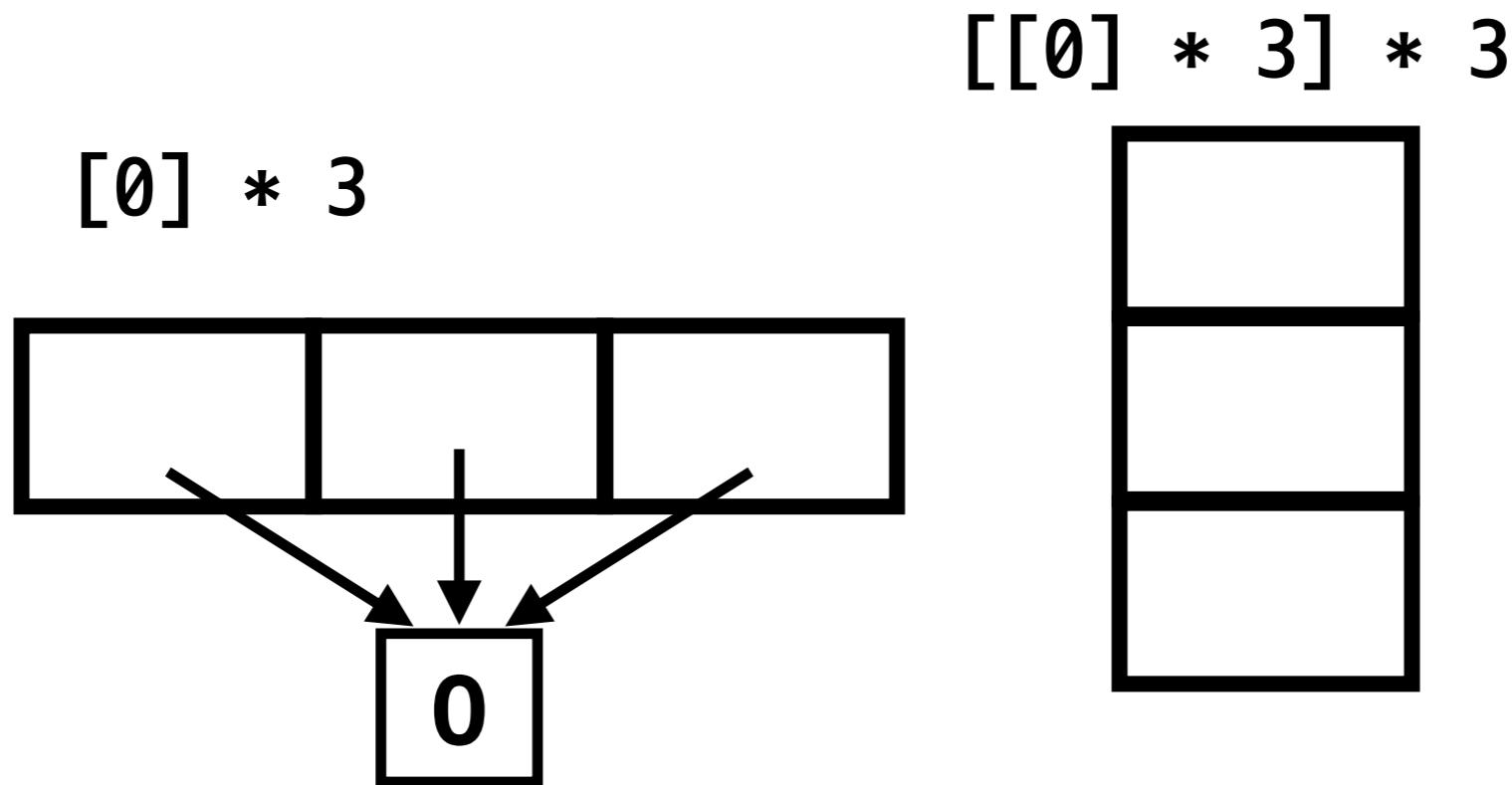
[[0] * 3] * 3

[0] * 3



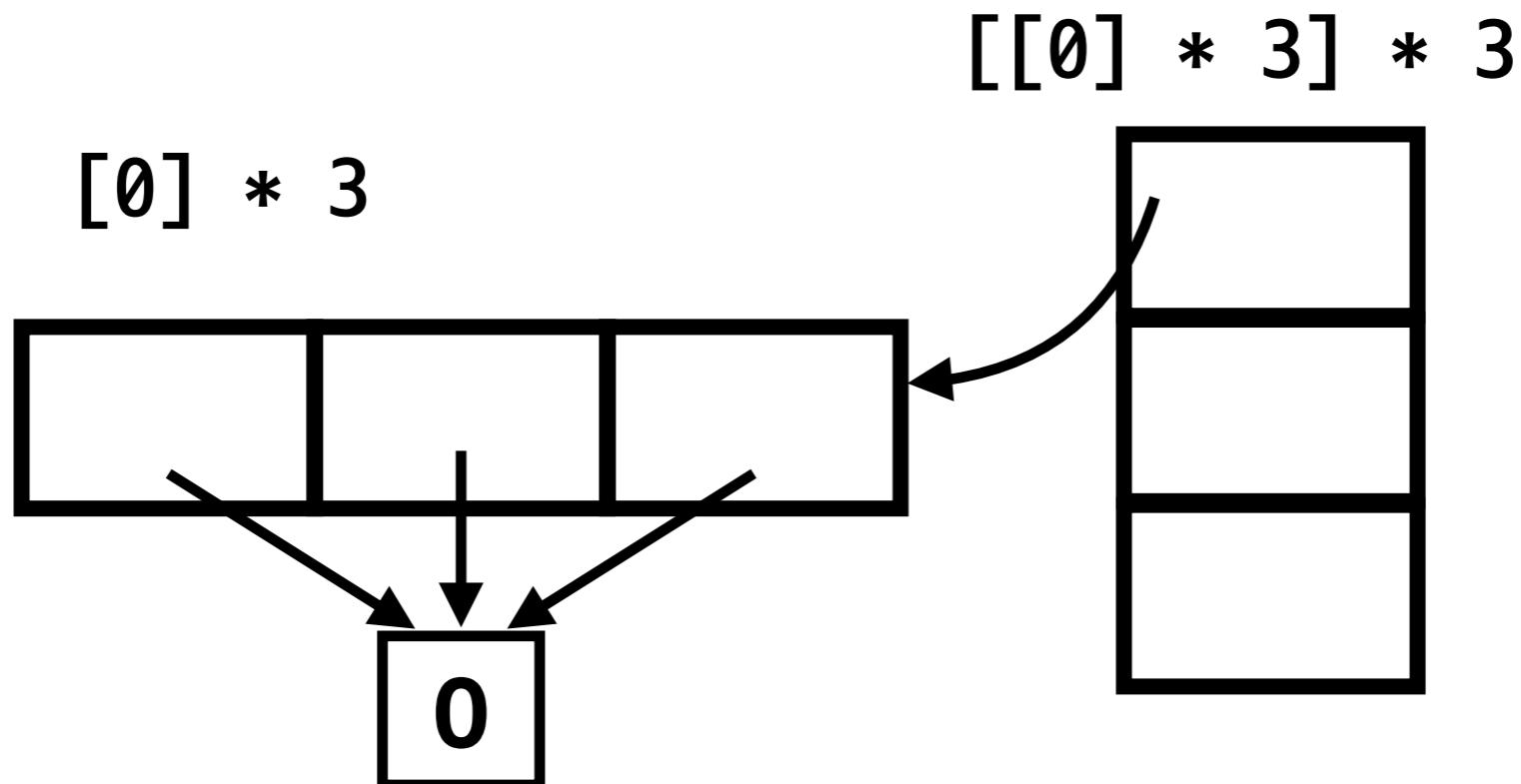
어떤 차이점?

board = [[0] * 3] * 3



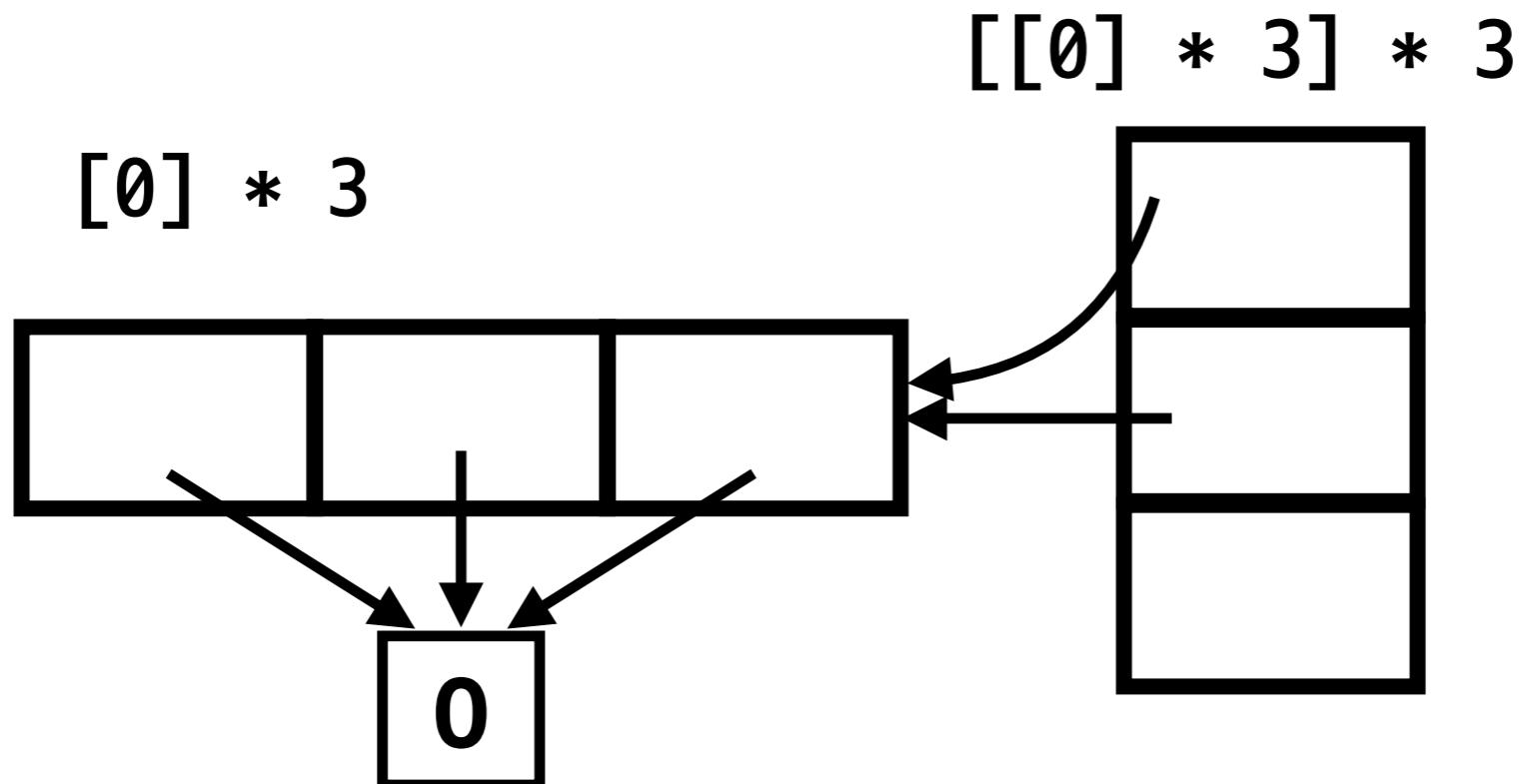
어떤 차이점?

board = [[0] * 3] * 3



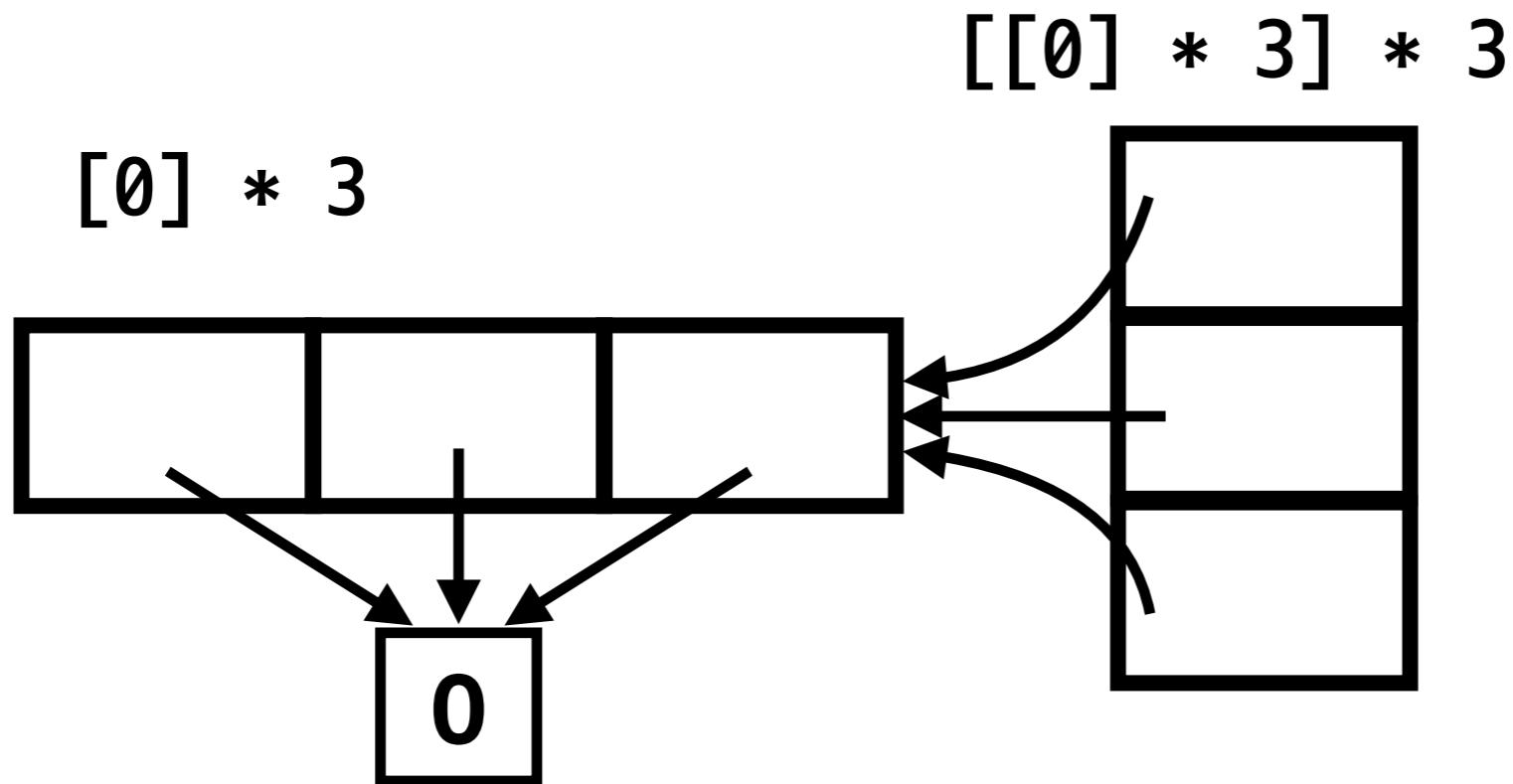
어떤 차이점?

board = [[0] * 3] * 3



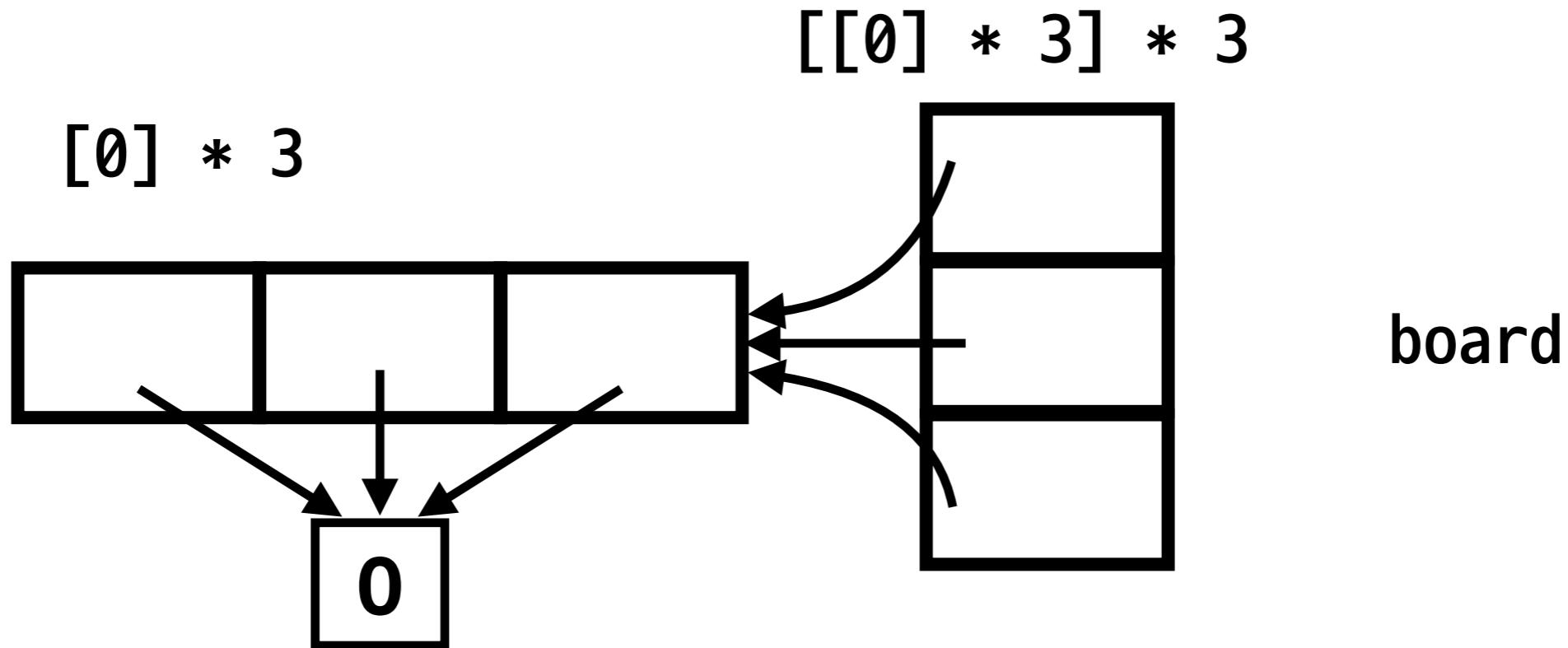
어떤 차이점?

board = [[0] * 3] * 3



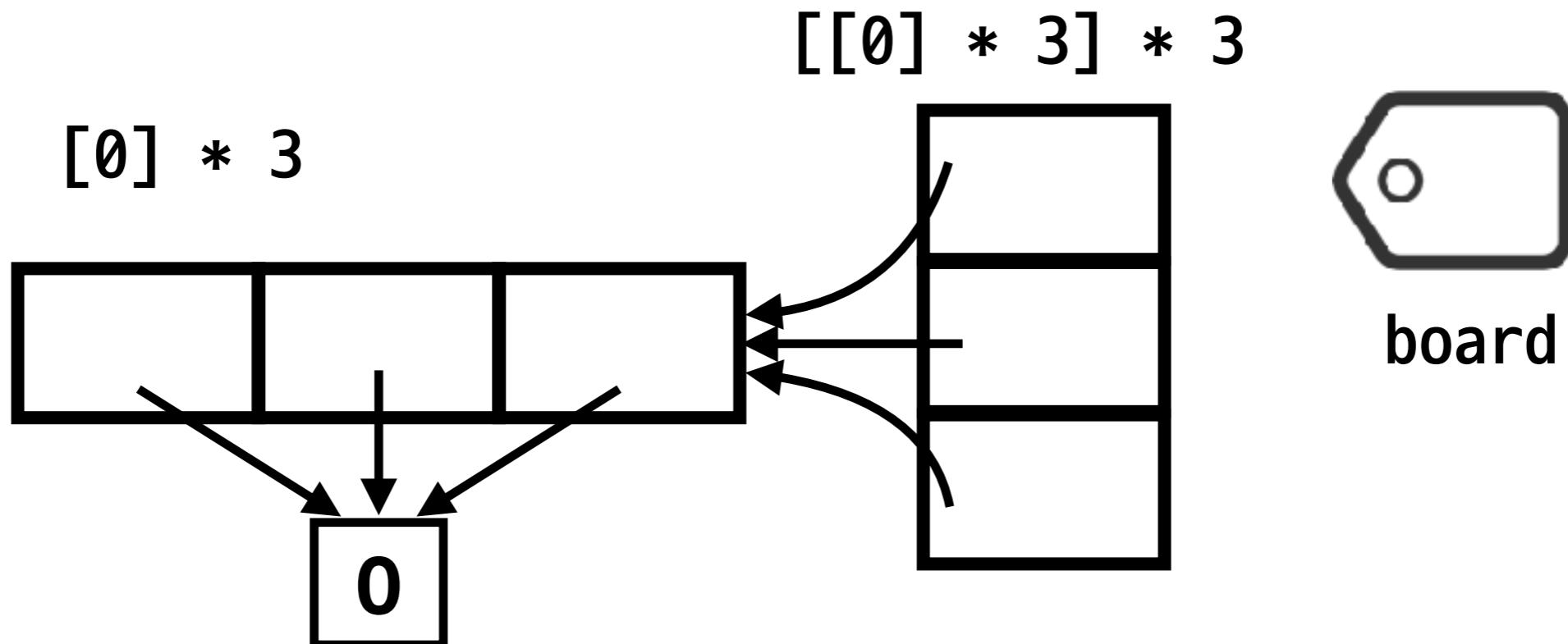
어떤 차이점?

board = [[0] * 3] * 3



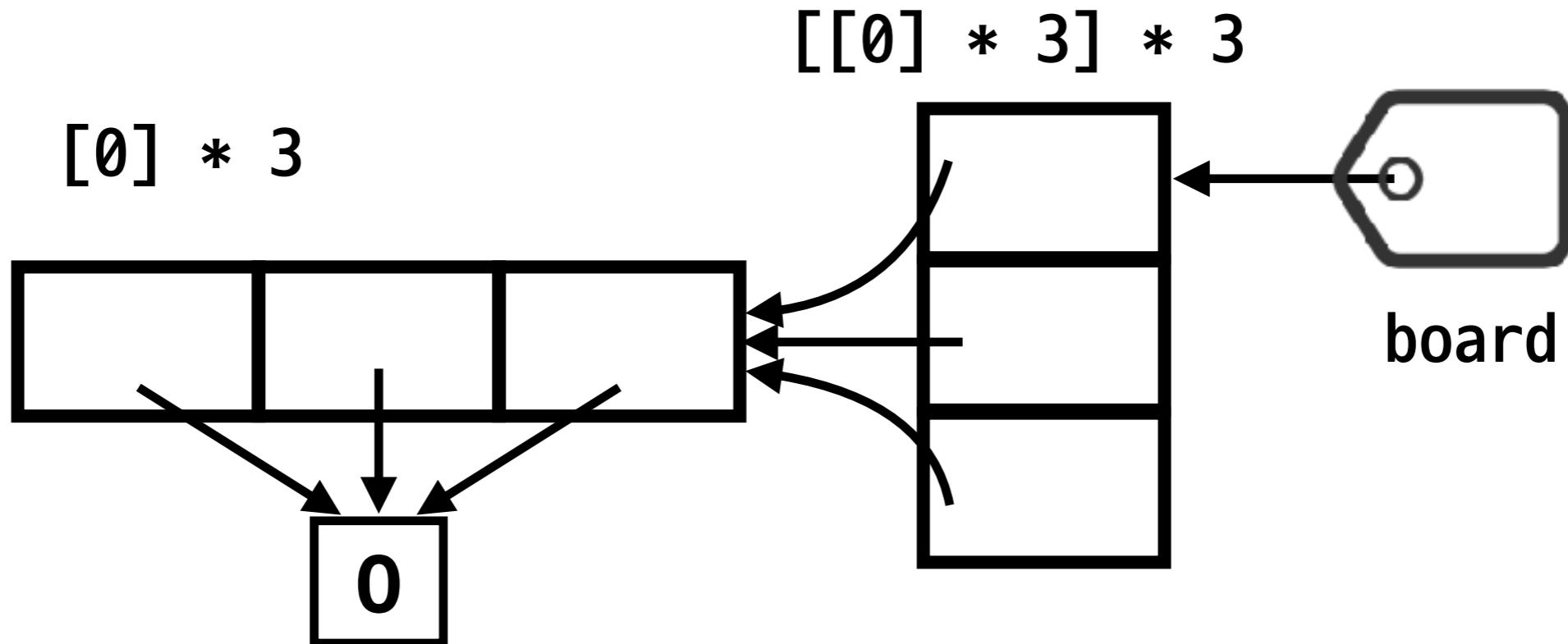
어떤 차이점?

board = [[0] * 3] * 3



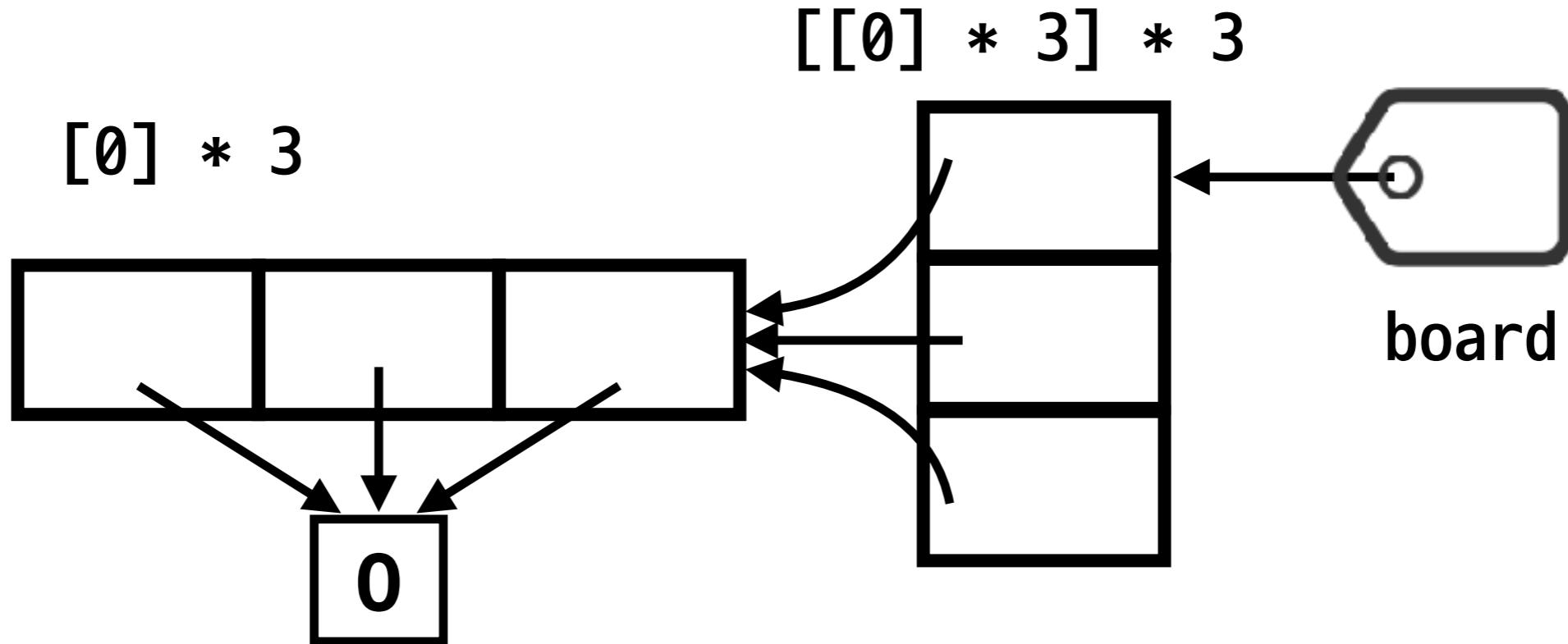
어떤 차이점?

board = [[0] * 3] * 3



어떤 차이점?

board = [[0] * 3] * 3

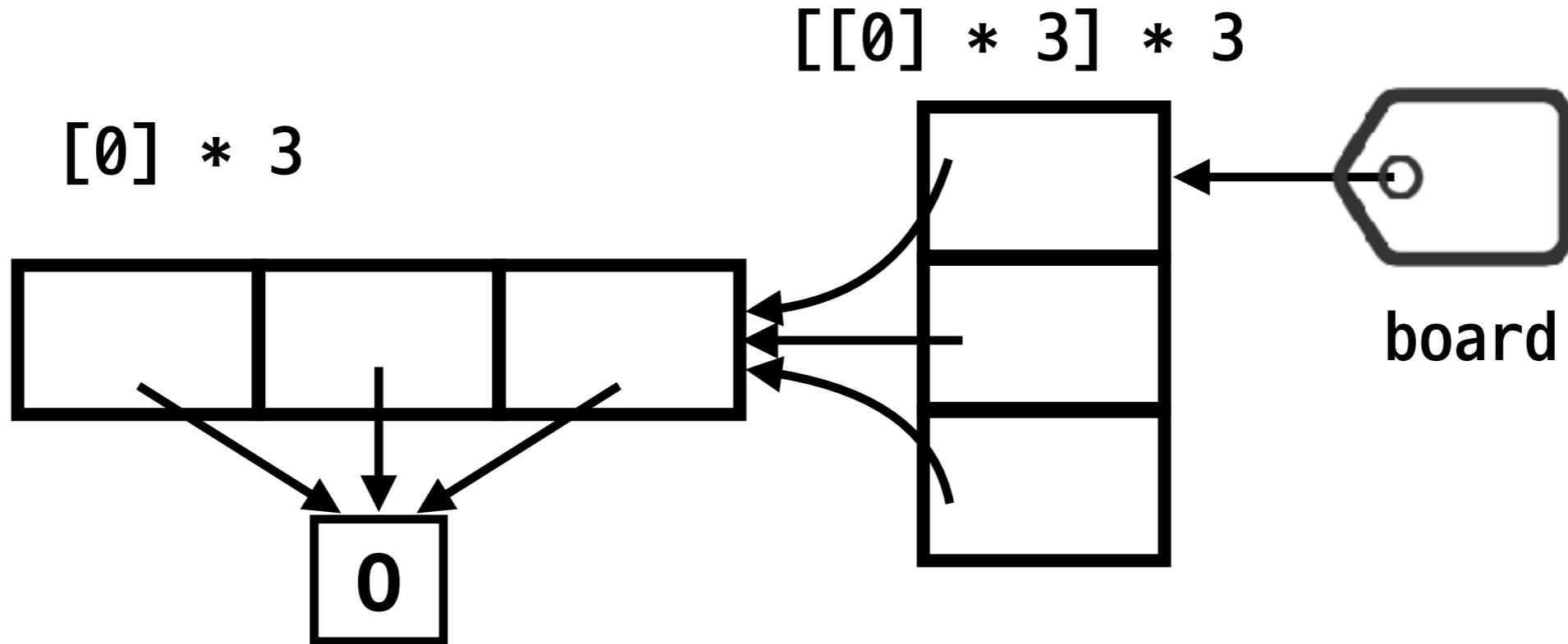


0 객체에 대한 3개의 참조

0객체에 대한 3개의 참조객체를 3개의 객체가 참조

어떤 차이점?

board = [[0] * 3] * 3



0 객체에 대한 3개의 참조

0객체에 대한 3개의 참조객체를 3개의 객체가 참조

어떤 차이점?

```
board = [[0] * 3 for _ in range(3)]
```

어떤 차이점?

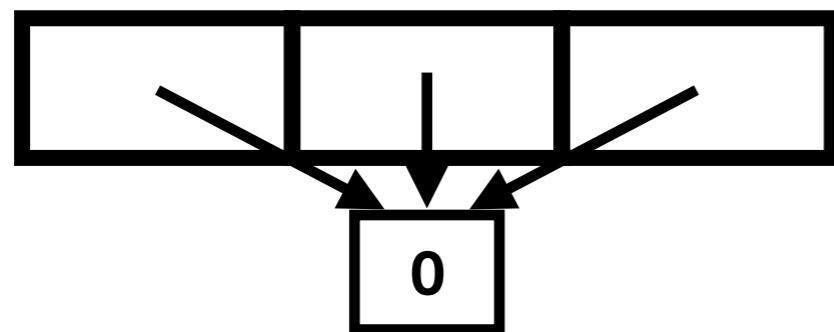
```
board = [[0] * 3 for _ in range(3)]
```

```
[0] * 3
```

어떤 차이점?

```
board = [[0] * 3 for _ in range(3)]
```

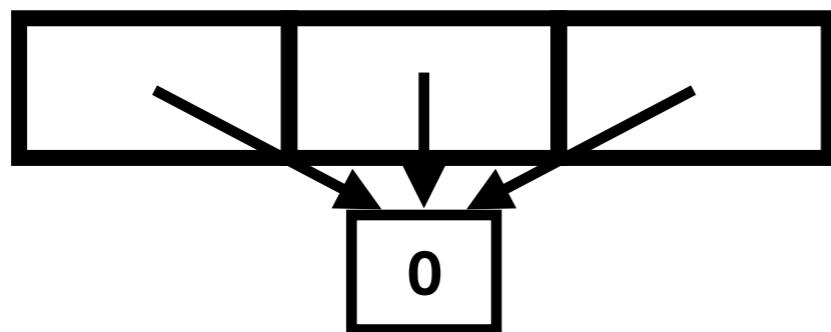
[0] * 3



어떤 차이점?

```
board = [[0] * 3 for _ in range(3)]
```

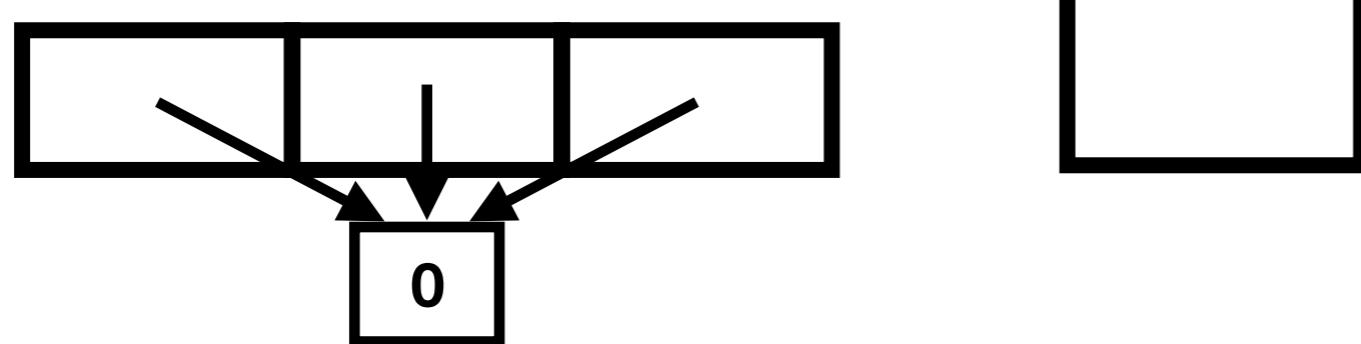
[0] * 3 [.. for _ in range(3)]



어떤 차이점?

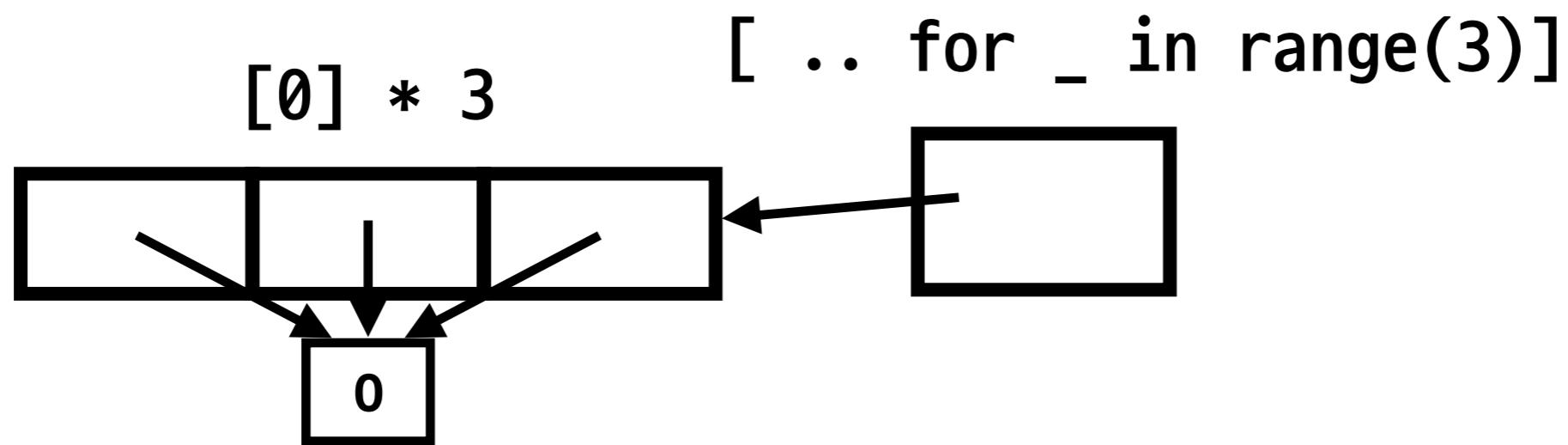
```
board = [[0] * 3 for _ in range(3)]
```

[0] * 3 [.. for _ in range(3)]



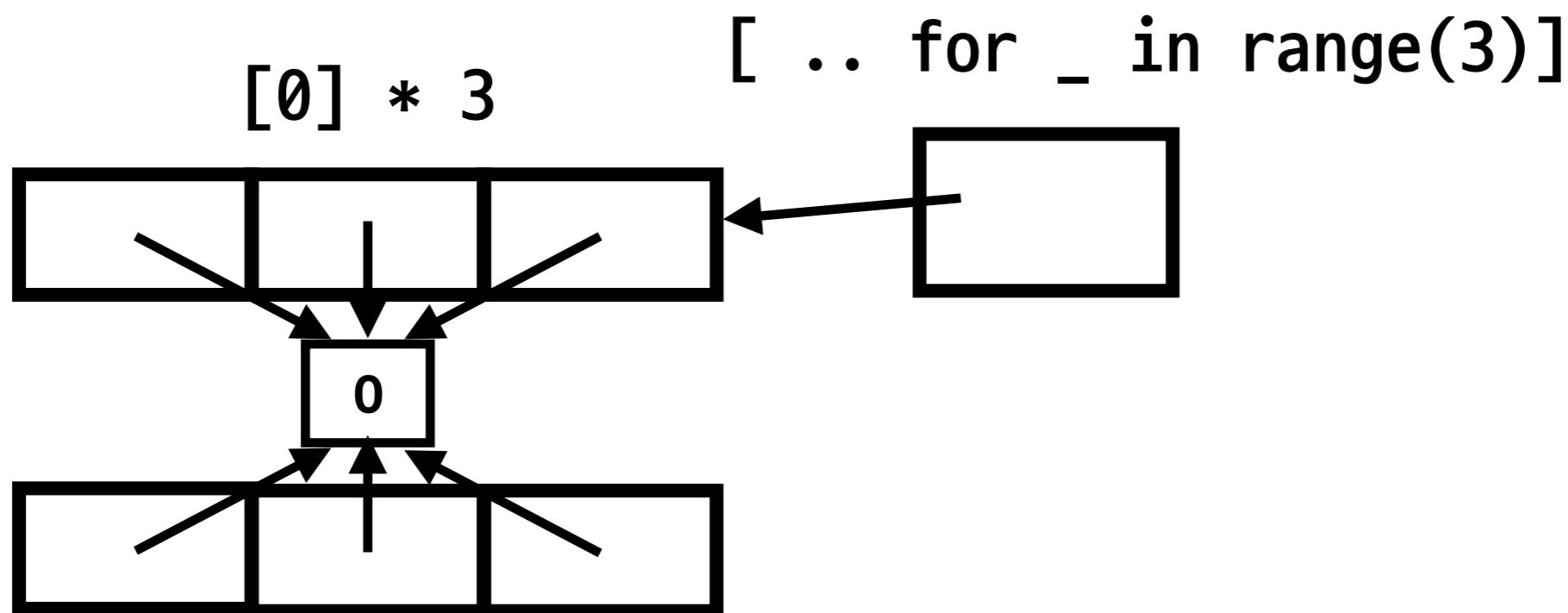
어떤 차이점?

```
board = [[0] * 3 for _ in range(3)]
```



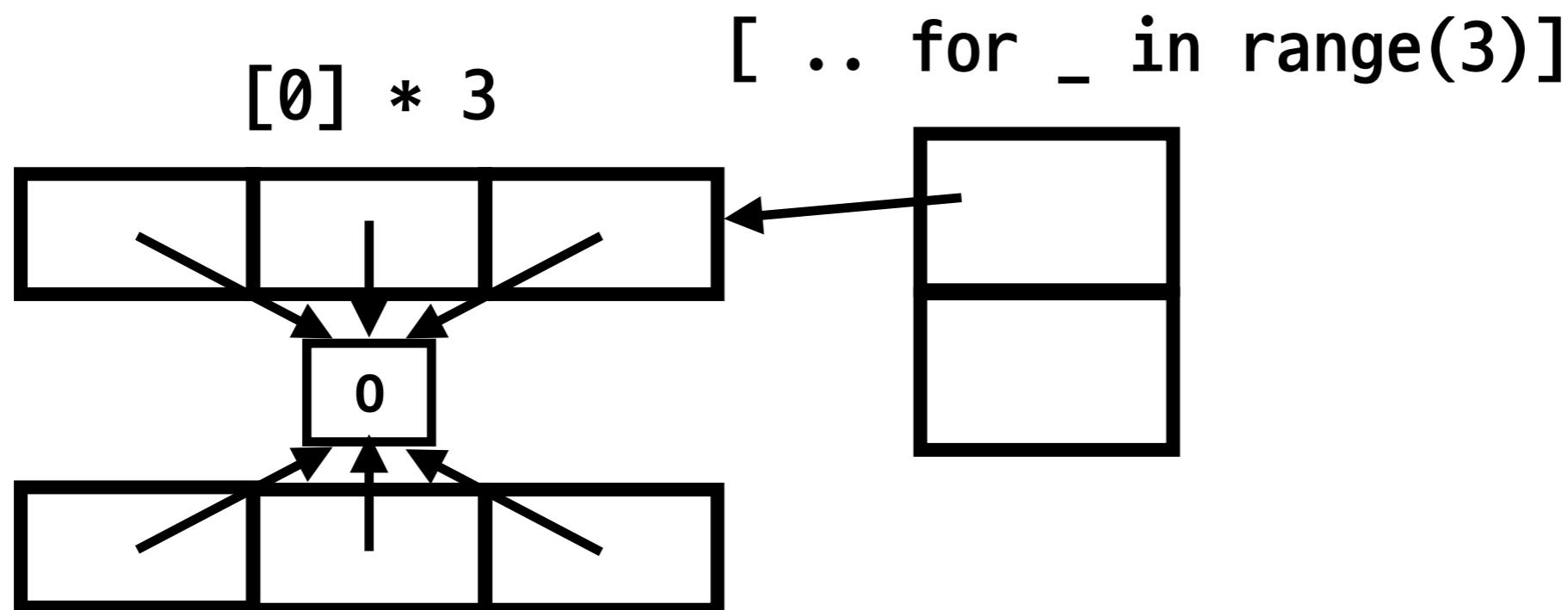
어떤 차이점?

```
board = [[0] * 3 for _ in range(3)]
```



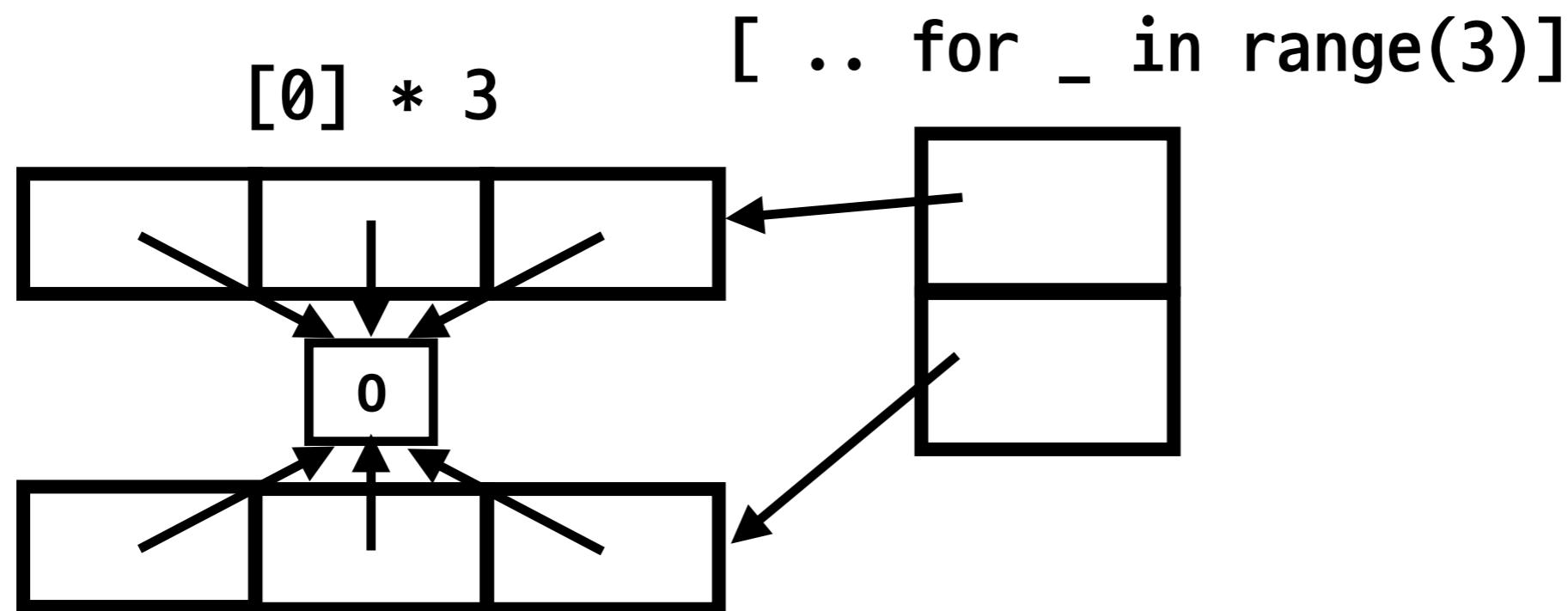
어떤 차이점?

```
board = [[0] * 3 for _ in range(3)]
```



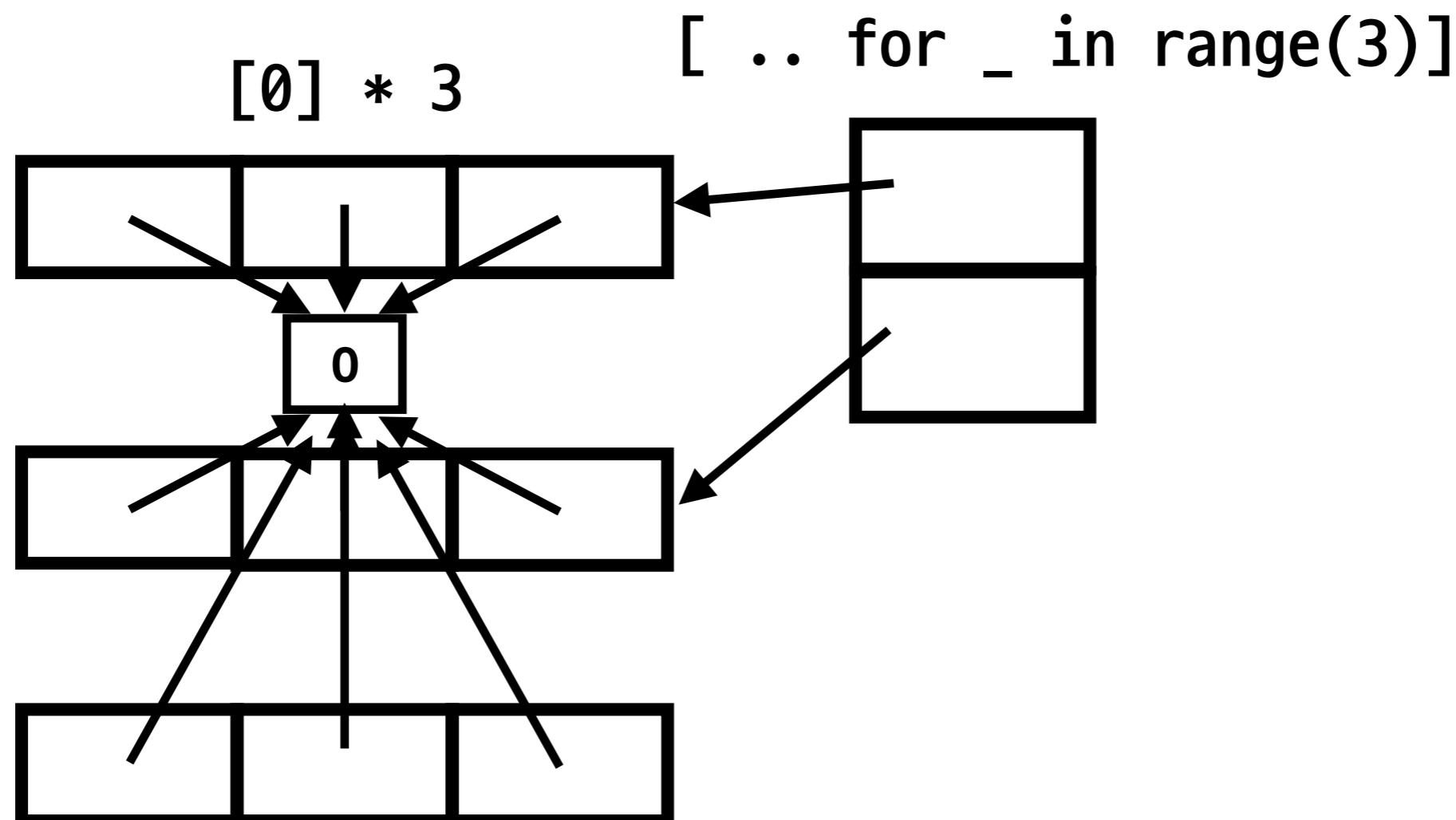
어떤 차이점?

```
board = [[0] * 3 for _ in range(3)]
```



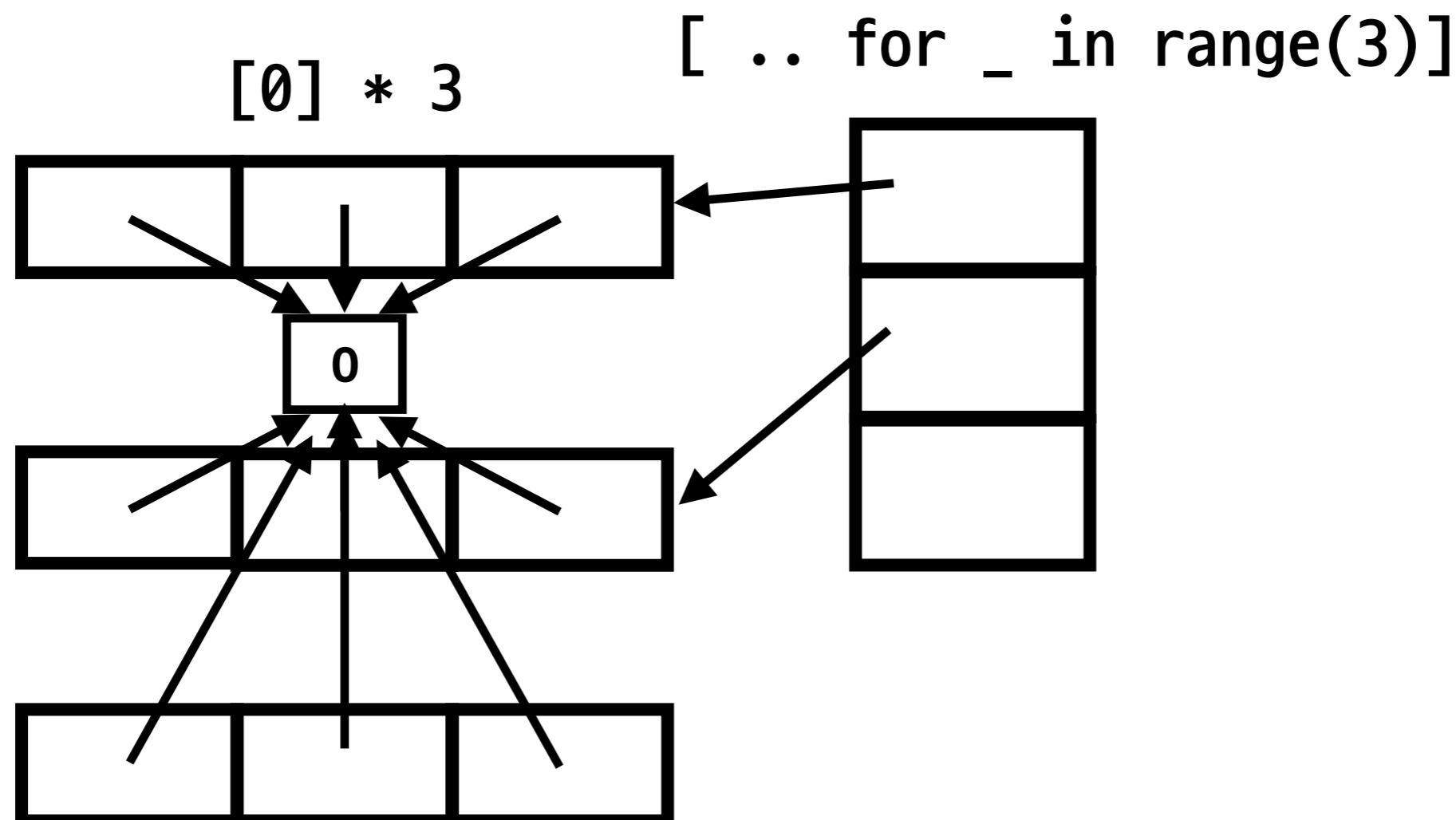
어떤 차이점?

```
board = [[0] * 3 for _ in range(3)]
```



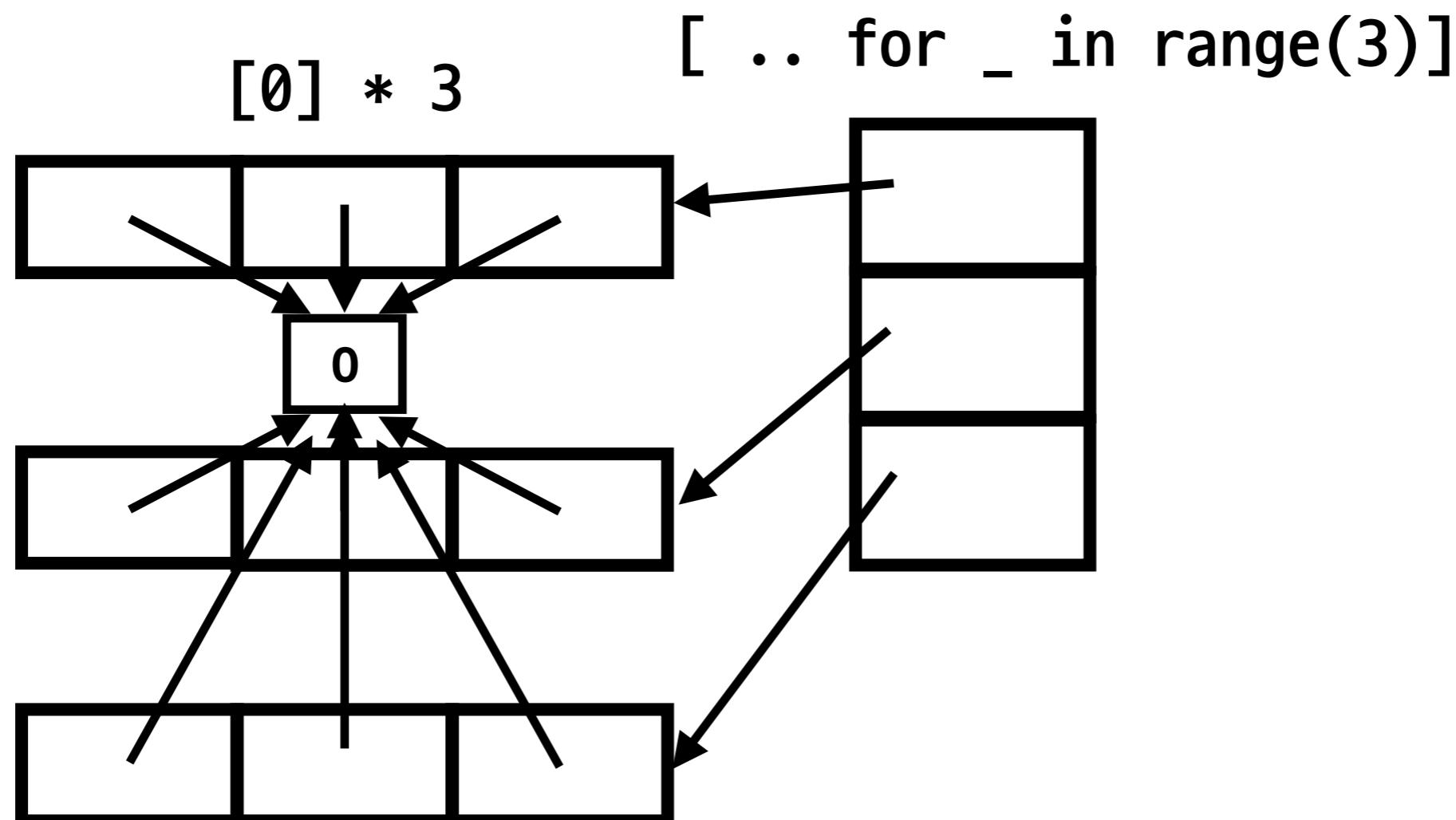
어떤 차이점?

```
board = [[0] * 3 for _ in range(3)]
```



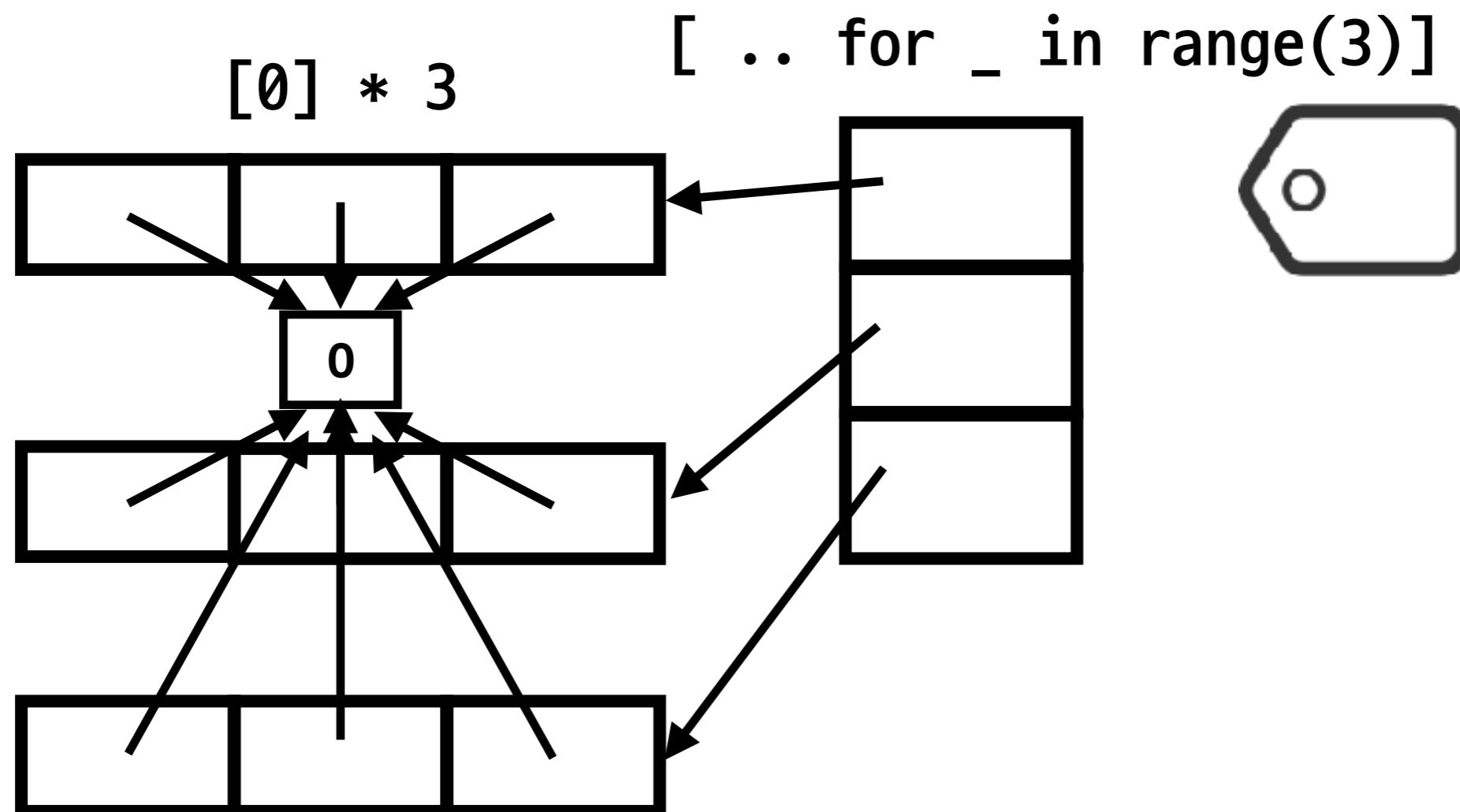
어떤 차이점?

```
board = [[0] * 3 for _ in range(3)]
```



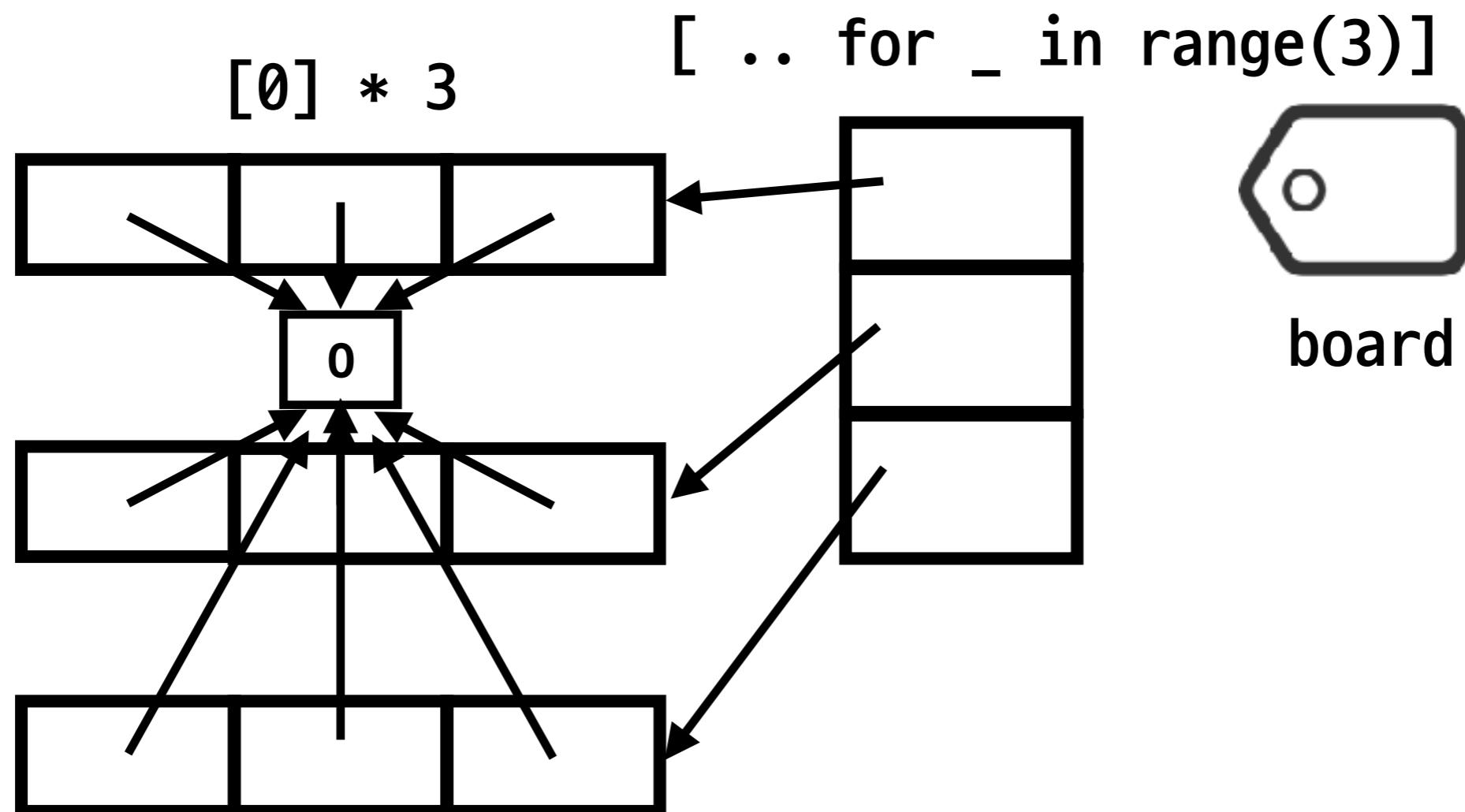
어떤 차이점?

```
board = [[0] * 3 for _ in range(3)]
```



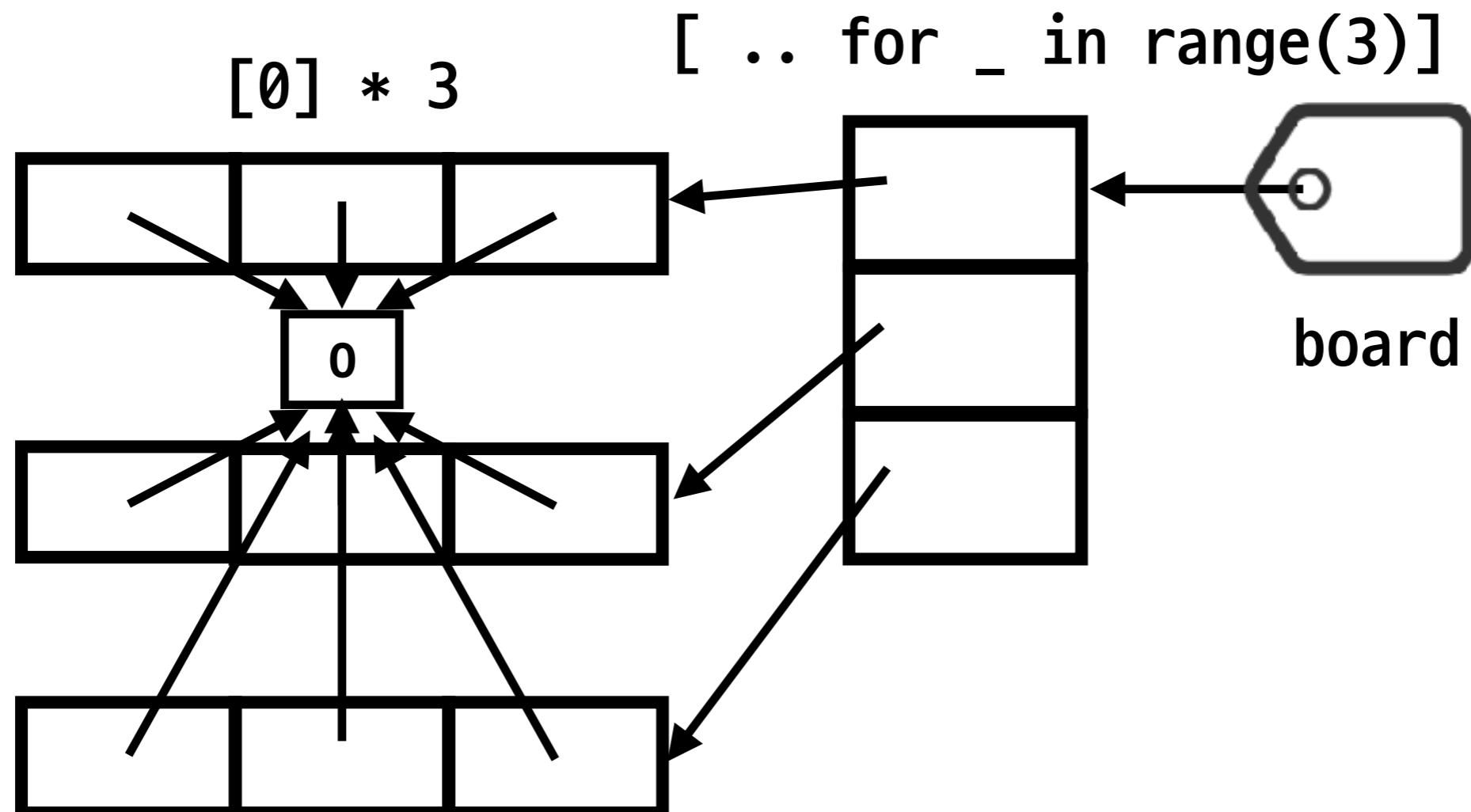
어떤 차이점?

```
board = [[0] * 3 for _ in range(3)]
```



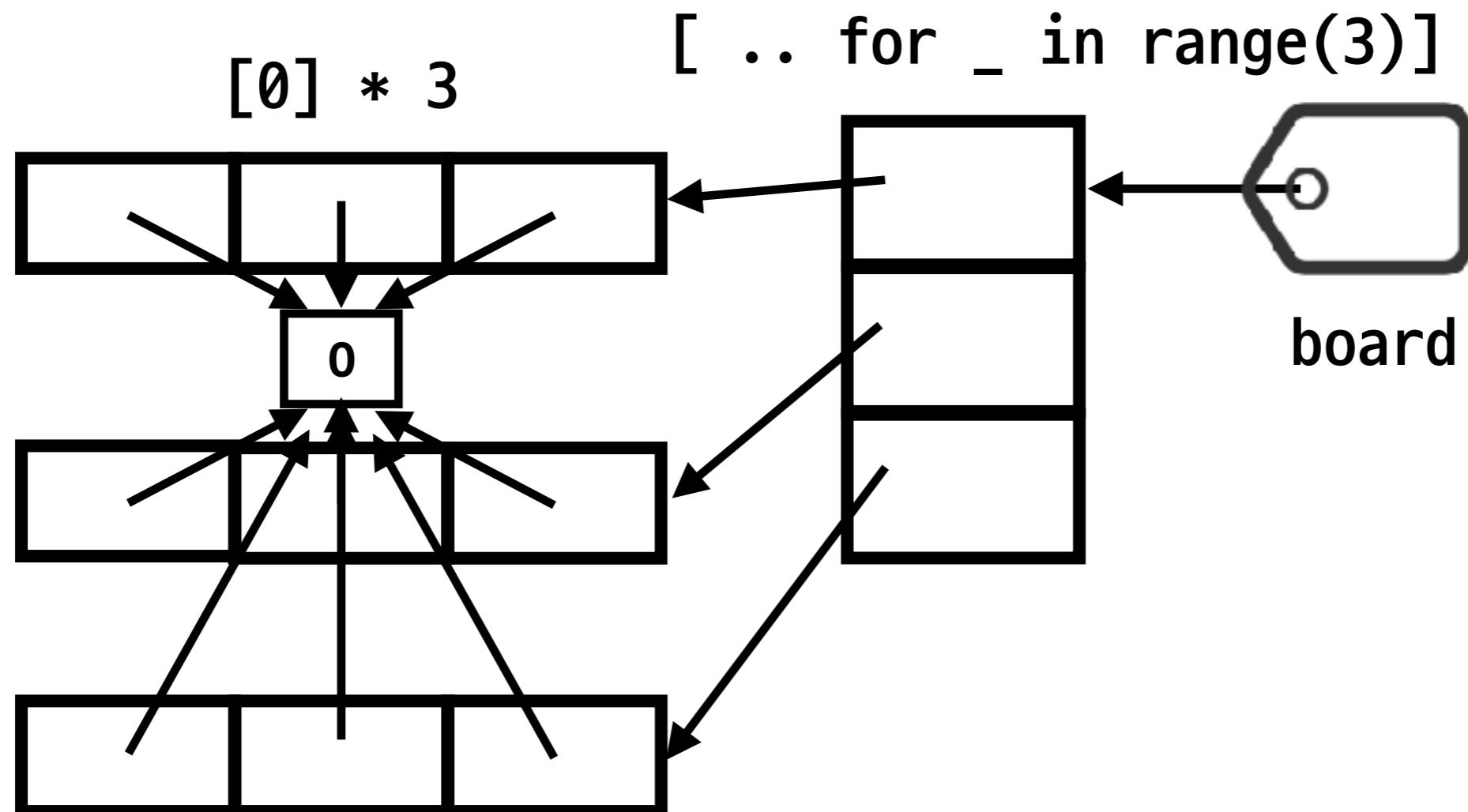
어떤 차이점?

```
board = [[0] * 3 for _ in range(3)]
```



어떤 차이점?

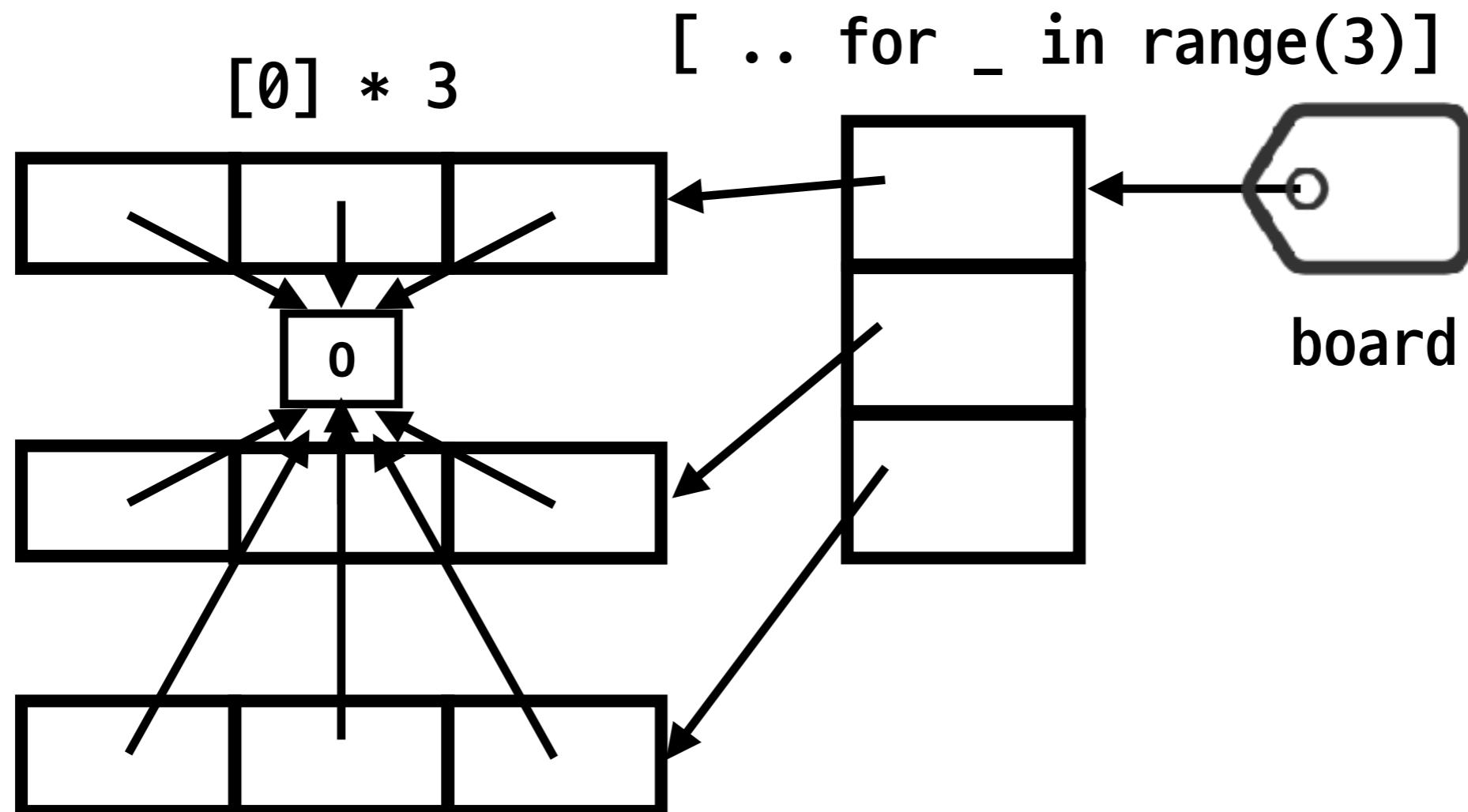
```
board = [[0] * 3 for _ in range(3)]
```



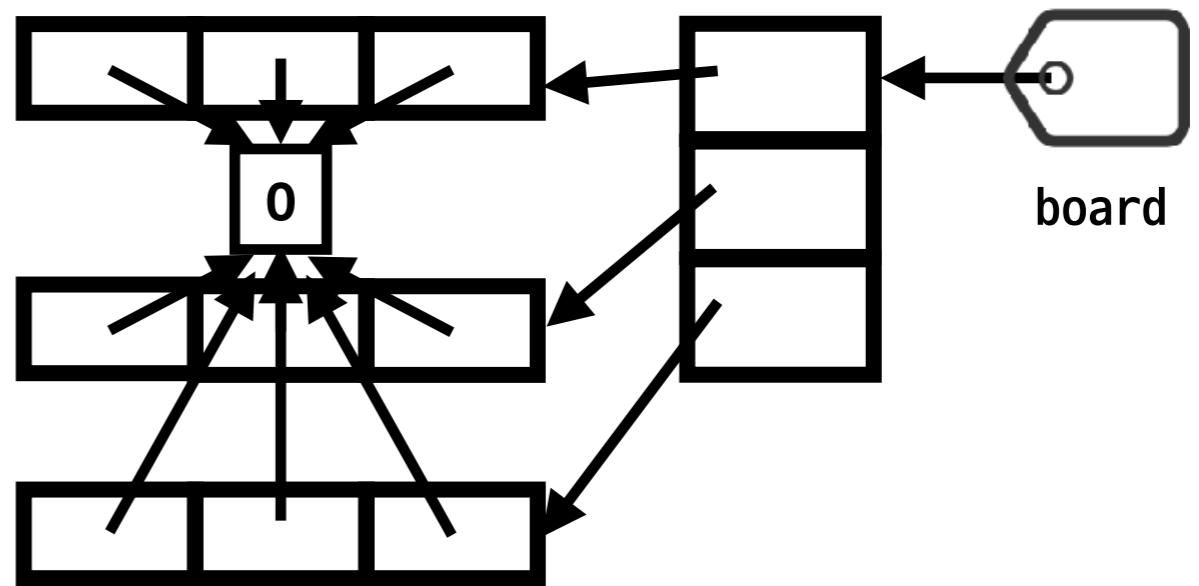
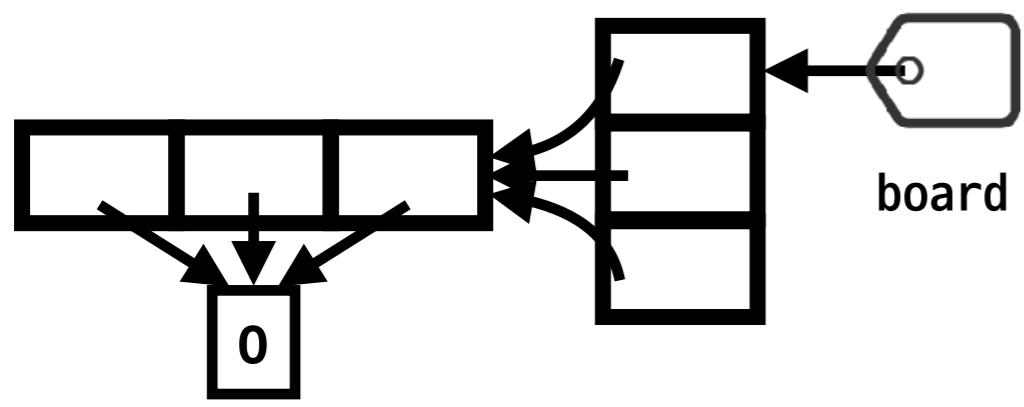
0 객체에 대한 3개의 참조
0객체에 대한 3개의 참조객체가 3개 생성되고
이를 참조하는 객체가 있더라

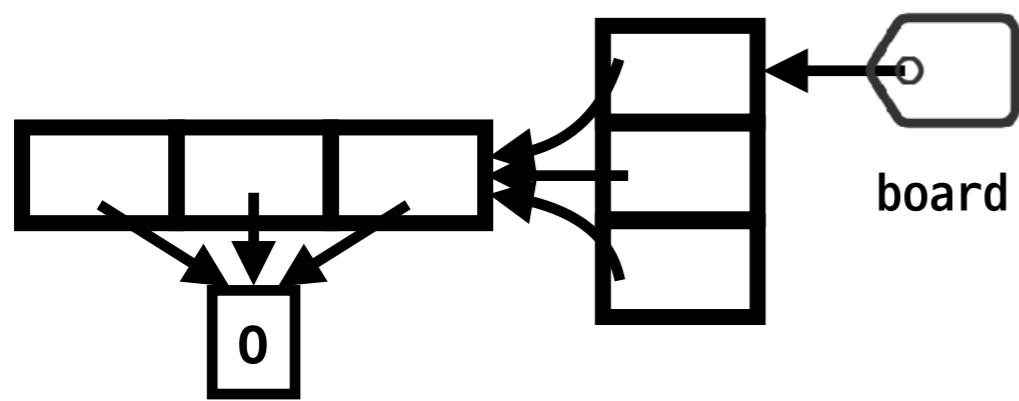
어떤 차이점?

```
board = [[0] * 3 for _ in range(3)]
```

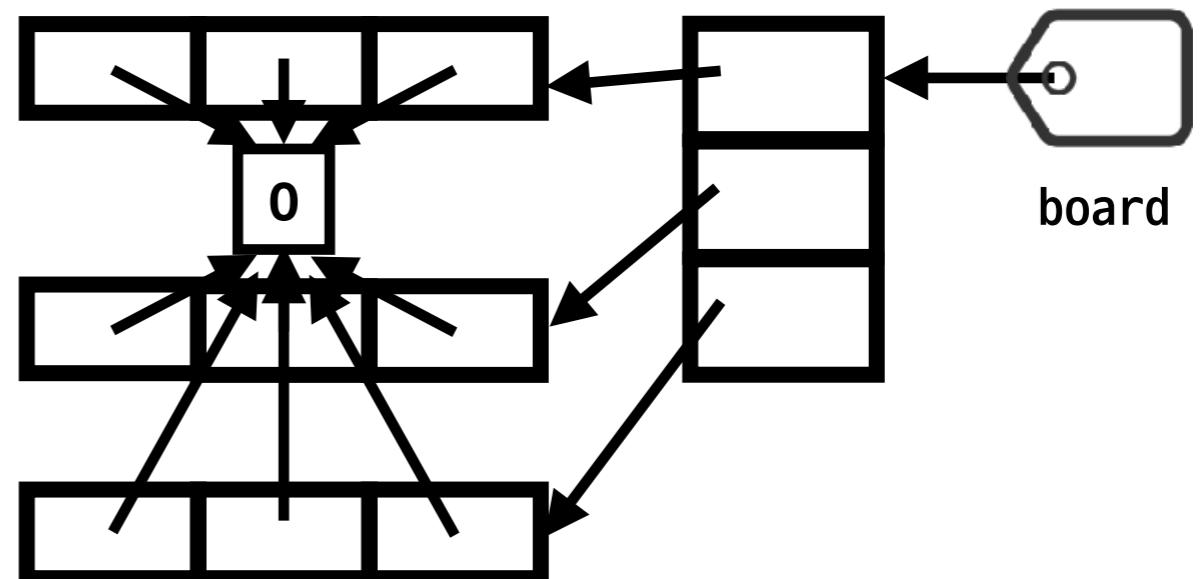


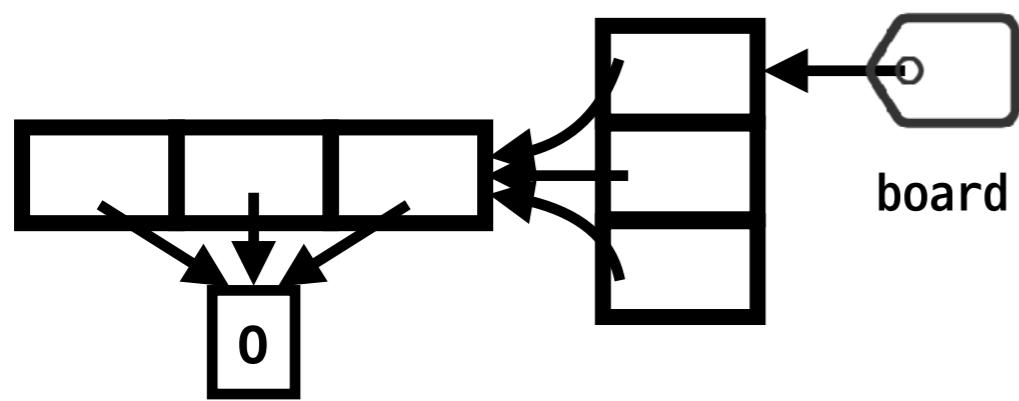
0 객체에 대한 3개의 참조
0객체에 대한 3개의 참조객체가 3개 생성되고
이를 참조하는 객체가 있더라



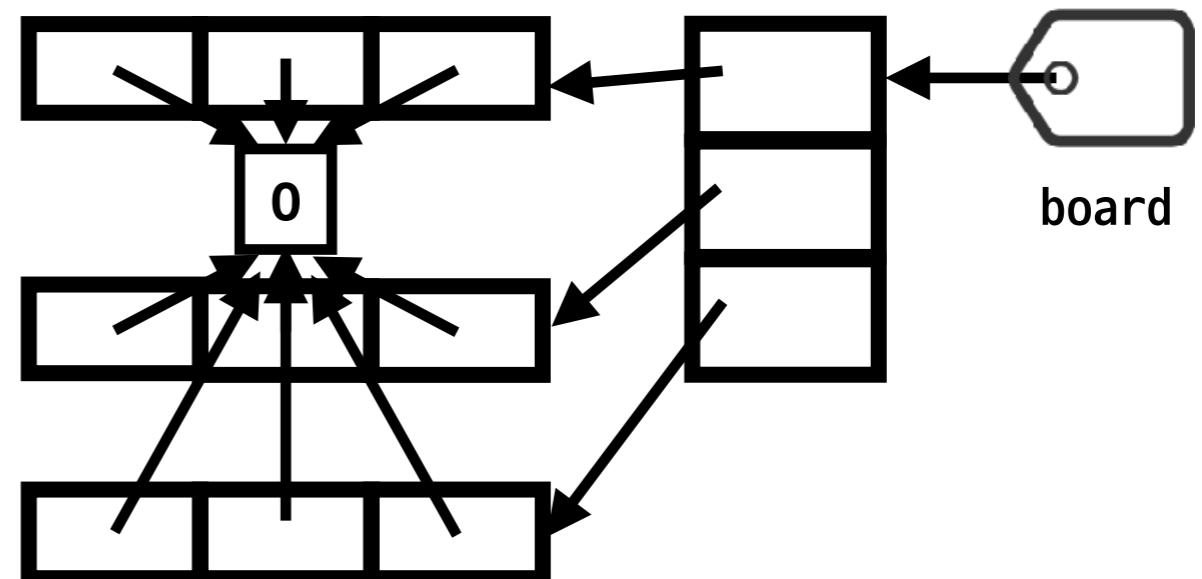


board = [[0] * cols] * rows

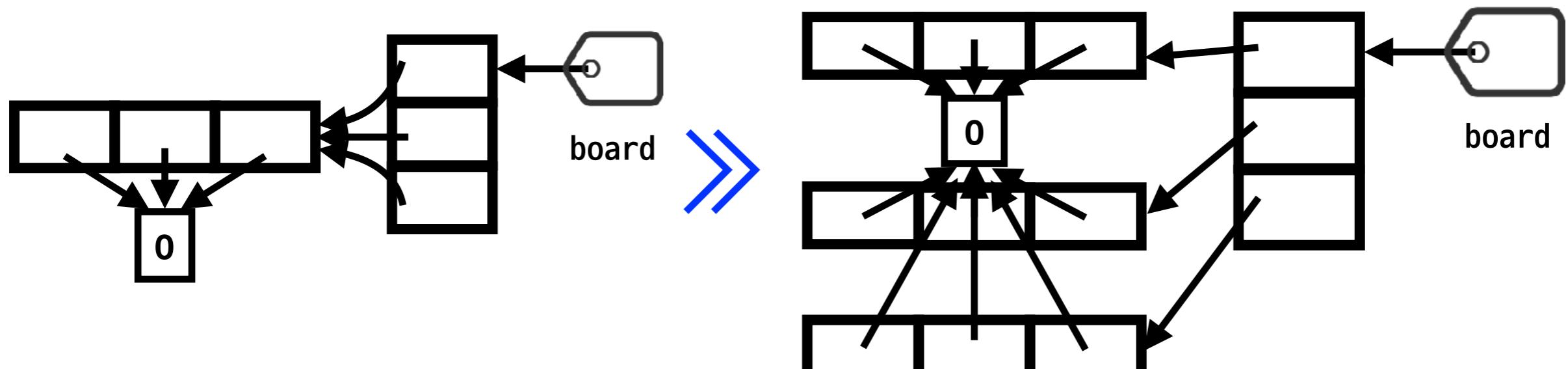




```
board = [[0] * cols] * rows
```

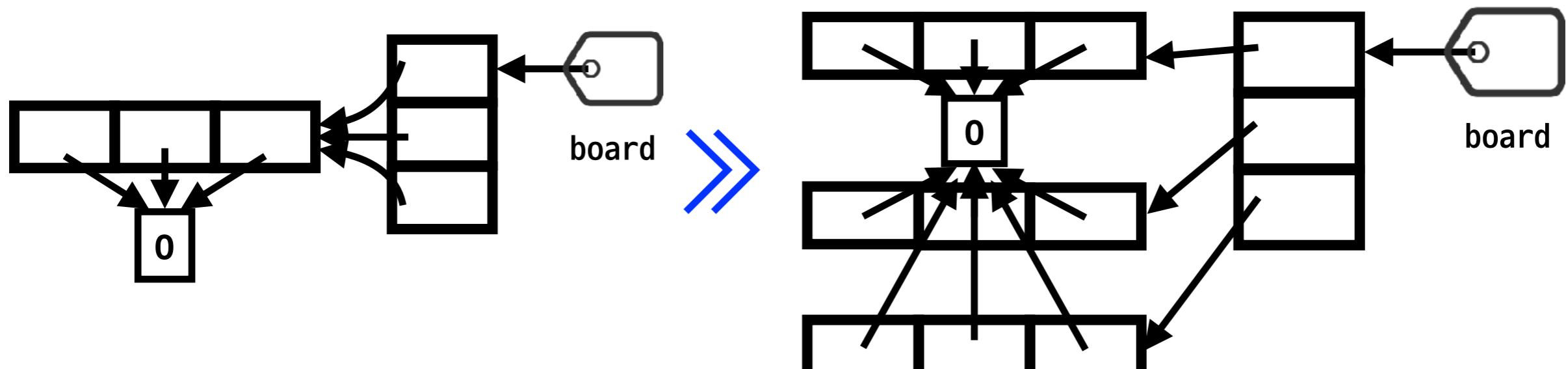


```
board = [[0] * cols for _ in range(rows)]
```



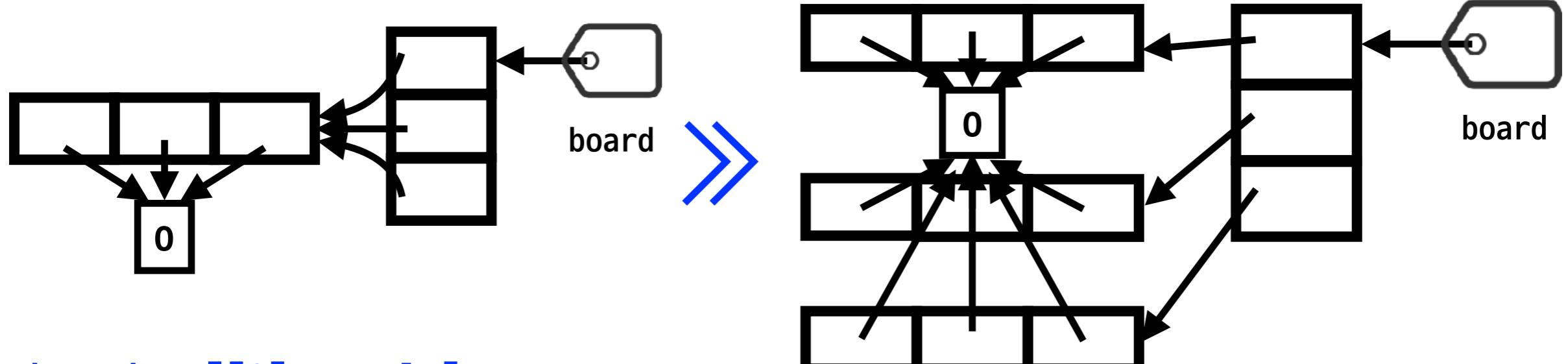
`board = [[0] * cols] * rows`

`board = [[0] * cols for _ in range(rows)]`



`board = [[0] * cols] * rows`

`board = [[0] * cols for _ in range(rows)]`



`board = [[0] * cols] * rows`

객체의 수도 적고
참조도 적게한다
= 속도가 빠른 이유

`board = [[0] * cols for _ in range(rows)]`

Why Bad?

```
1 board = [[0] * 3] * 3 # 나쁜 결과
2 print(board)
3 print(id(board[0])) # 아래 객체와 동일함
4 print(id(board[1]))
5
6 print(id(board[0][0]))
7 print(id(board[0][1]))
8 print(id(board[0][2]))
9 print(id(board[1][0]))
10 print(id(board[1][1]))
11 print(id(board[1][2]))
12
13 # 첫 항목 값만 고침
14 board[0][0] = 1
15 print(board)
```

	[[0, 0, 0], [0, 0, 0], [0, 0, 0]]
6	139869936803144
7	139869936803144
8	10910368
9	10910368
10	10910368
11	10910368
13	10910368
14	10910368
15	10910368
	[[1, 0, 0], [1, 0, 0], [1, 0, 0]]

Why Bad?

```
1 board = [[0] * 3] * 3 # 나쁜 결과
2 print(board)
3 print(id(board[0])) # 아래 객체와 동일함
4 print(id(board[1]))
5
6 print(id(board[0][0]))
7 print(id(board[0][1]))
8 print(id(board[0][2]))
9 print(id(board[1][0]))
10 print(id(board[1][1]))
11 print(id(board[1][2]))
12
13 # 첫 항목 값만 고침
14 board[0][0] = 1
15 print(board)
```

	[[0, 0, 0], [0, 0, 0], [0, 0, 0]]
13	139869936803144
14	139869936803144
15	10910368
16	10910368
17	10910368
18	10910368
19	10910368
20	10910368
21	[[1, 0, 0], [1, 0, 0], [1, 0, 0]]

Why Bad?

```
1 board = [[0] * 3] * 3 # 나쁜 결과
2 print(board)
3 print(id(board[0])) # 아래 객체와 동일함
4 print(id(board[1]))
5
6 print(id(board[0][0]))
7 print(id(board[0][1]))
8 print(id(board[0][2]))
9 print(id(board[1][0]))
10 print(id(board[1][1]))
11 print(id(board[1][2]))
12
13 # 첫 항목 값만 고침
14 board[0][0] = 1
15 print(board)
```

	[[0, 0, 0], [0, 0, 0], [0, 0, 0]]
	139869936803144
	139869936803144
	10910368
	10910368
	10910368
	10910368
	10910368
	10910368
	10910368
	[[1, 0, 0], [1, 0, 0], [1, 0, 0]]

Why Good?

```
1 board = [[0] * 3 for _ in range(3)]
2 print(board)
3 print(id(board[0]))          # 아래 객체와 다름
4 print(id(board[1]))
5 print(id(board[0][0]))
6 print(id(board[0][1]))
7 print(id(board[0][2]))
8 print(id(board[1][0]))
9 print(id(board[1][1]))
10 print(id(board[1][2]))
11
12 # 첫 항목 값만 고침
13 board[0][0] = 1
14 print(board)
```

	[[0, 0, 0], [0, 0, 0], [0, 0, 0]]
13	139869936802568
13	139869936802888
13	10910368
13	10910368
13	10910368
13	10910368
13	10910368
	[[1, 0, 0], [0, 0, 0], [0, 0, 0]]

Why Good?

```
1 board = [[0] * 3 for _ in range(3)]
2 print(board)
3 print(id(board[0]))          # 아래 객체와 다름
4 print(id(board[1]))
5 print(id(board[0][0]))        [[0, 0, 0], [0, 0, 0], [0, 0, 0]]
6 print(id(board[0][1]))        139869936802568
7 print(id(board[0][2]))        139869936802888
8 print(id(board[1][0]))        10910368
9 print(id(board[1][1]))        10910368
10 print(id(board[1][2]))       10910368
11
12 # 첫 항목 값만 고침
13 board[0][0] = 1
14 print(board)                [[1, 0, 0], [0, 0, 0], [0, 0, 0]]
```

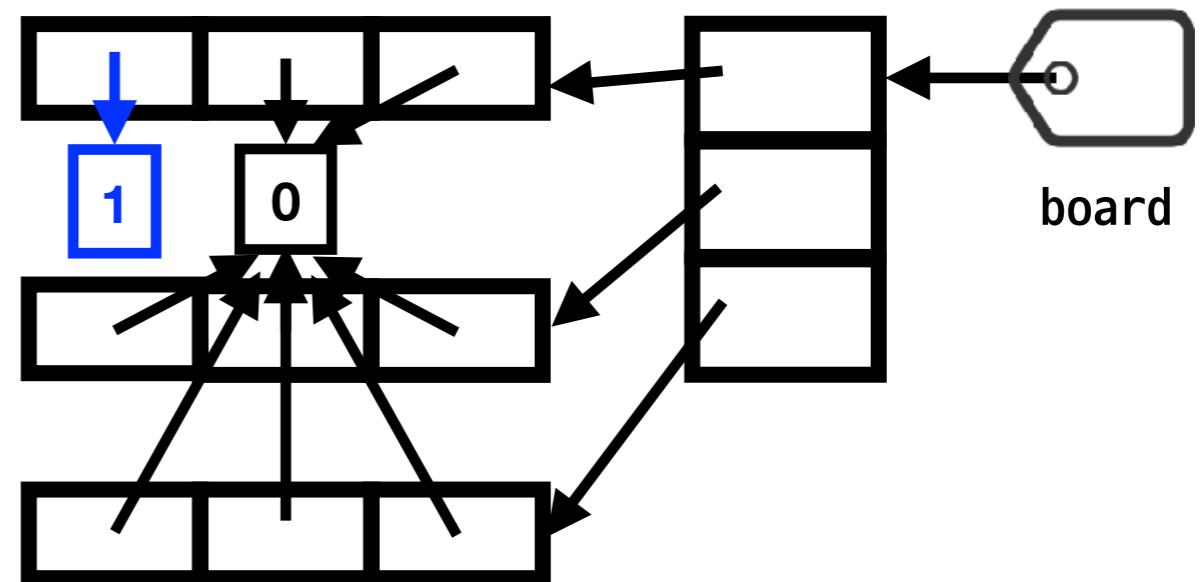
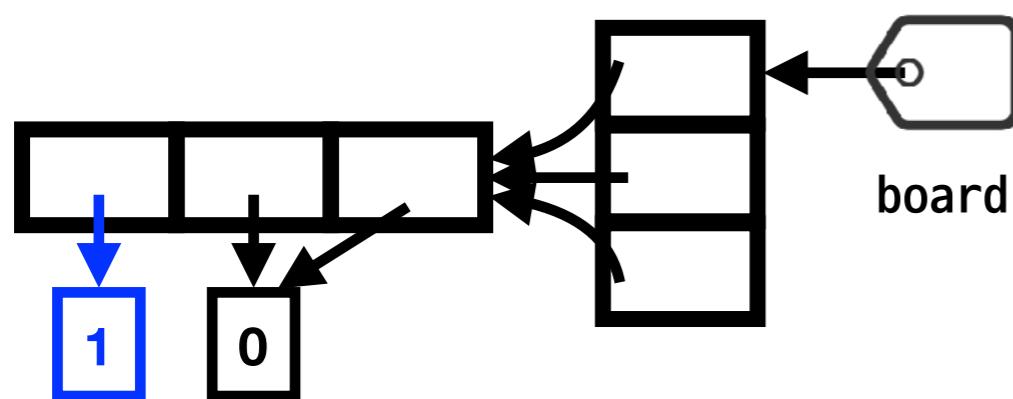
Why Good?

```
1 board = [[0] * 3 for _ in range(3)]
2 print(board)
3 print(id(board[0]))          # 아래 객체와 다름
4 print(id(board[1]))
5 print(id(board[0][0]))        [[0, 0, 0], [0, 0, 0], [0, 0, 0]]
6 print(id(board[0][1]))        139869936802568
7 print(id(board[0][2]))        139869936802888
8 print(id(board[1][0]))        10910368
9 print(id(board[1][1]))        10910368
10 print(id(board[1][2]))       10910368
11
12 # 첫 항목 값만 고침
13 board[0][0] = 1
14 print(board)                [[1, 0, 0], [0, 0, 0], [0, 0, 0]]
```

board[0][0] = 1 결과

board = [[0] * cols] * rows

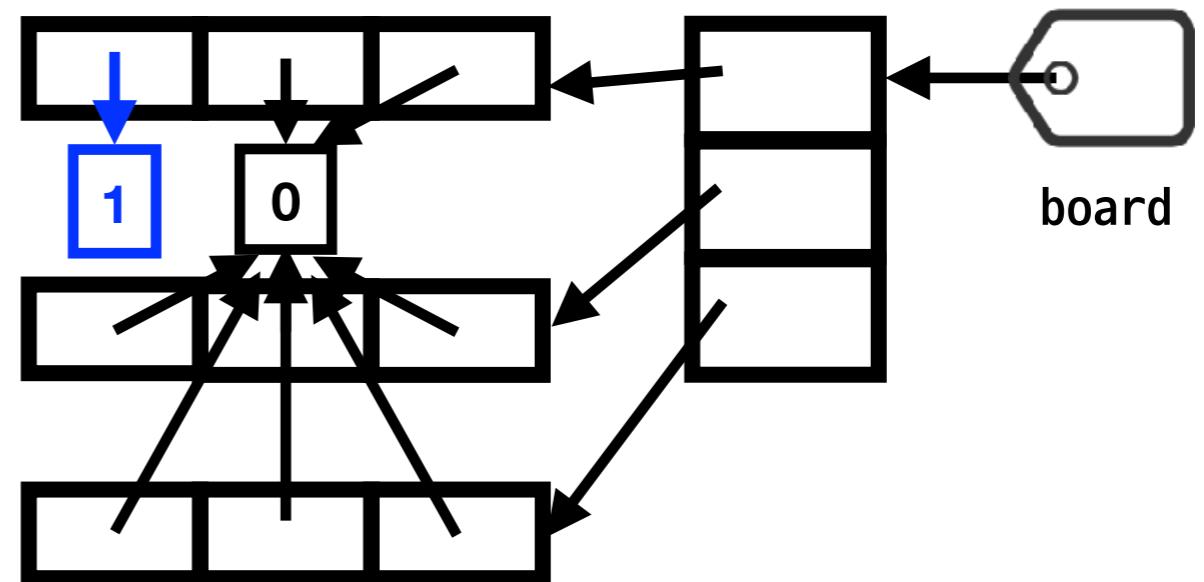
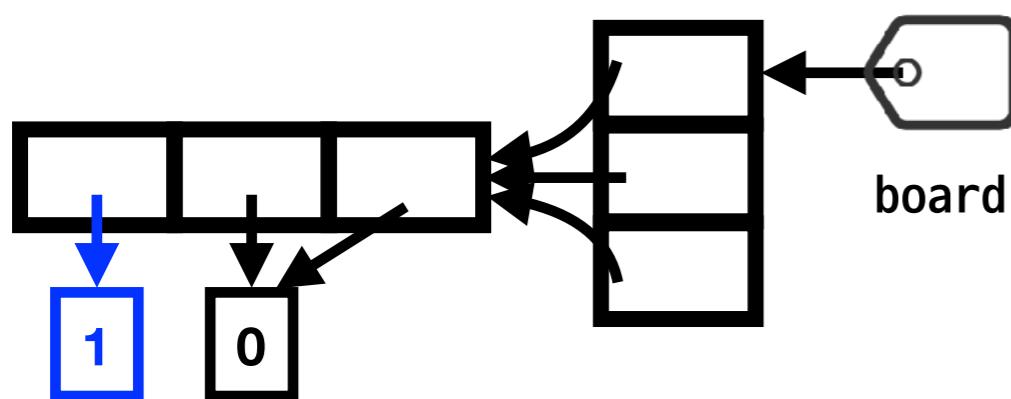
board = [[0] * cols for _ in range(rows)]



board[0][0] = 1 결과

board = [[0] * cols] * rows

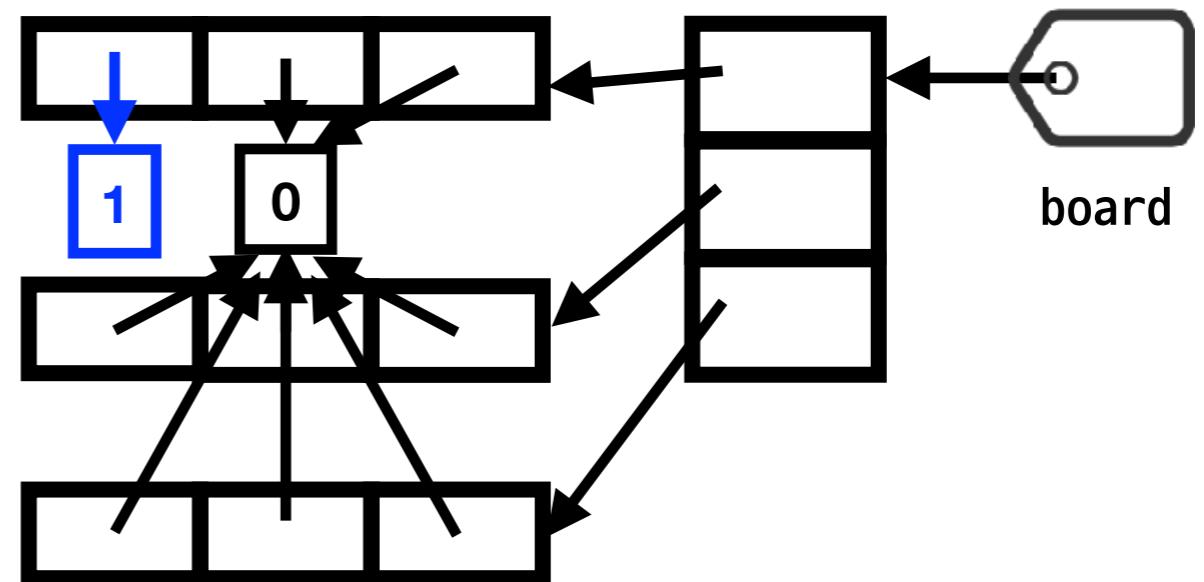
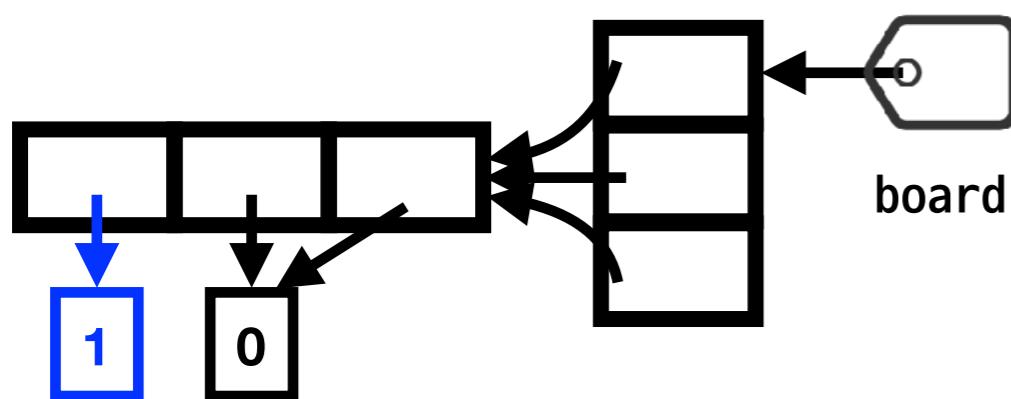
board = [[0] * cols for _ in range(rows)]



board[0][0] = 1 결과

board = [[0] * cols] * rows

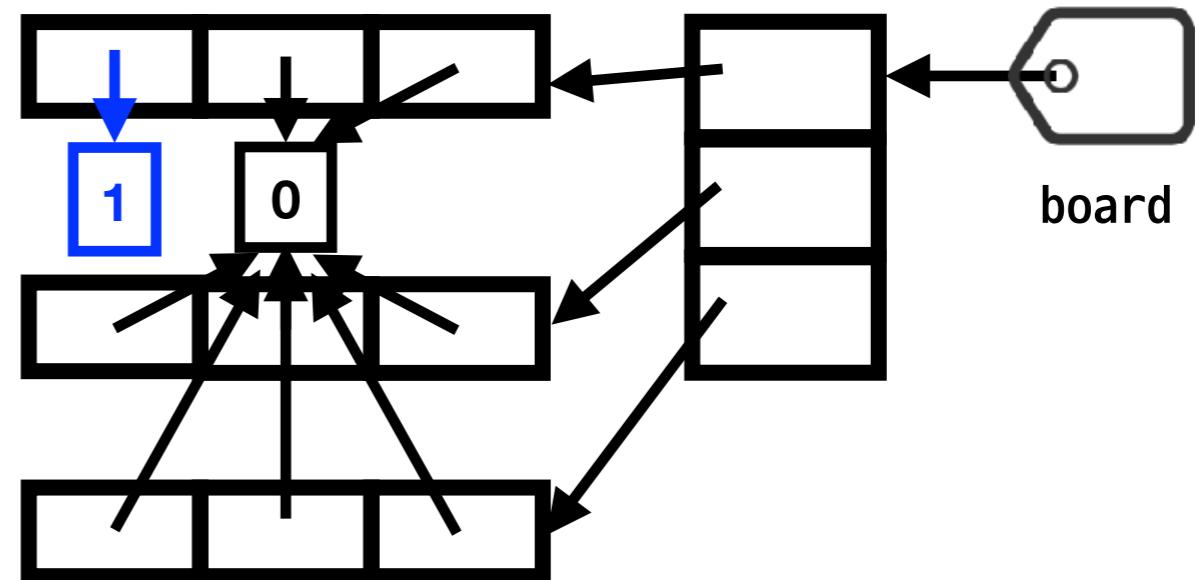
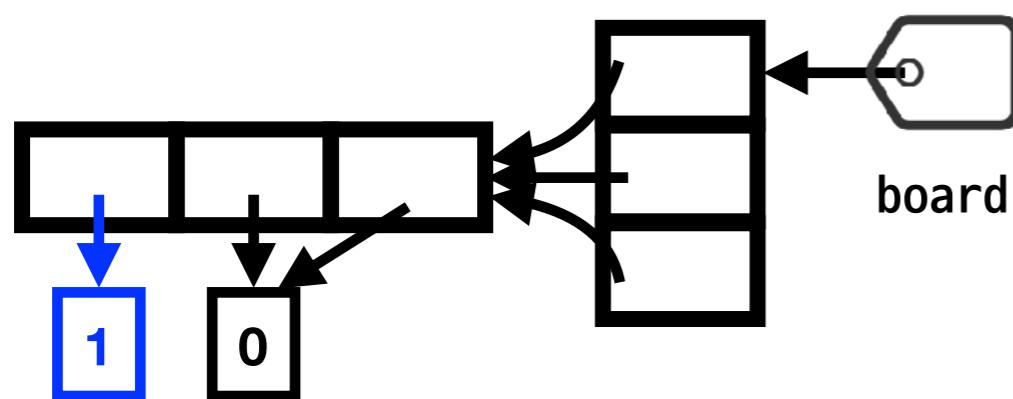
board = [[0] * cols for _ in range(rows)]



board[0][0] = 1 결과

board = [[0] * cols] * rows

board = [[0] * cols for _ in range(rows)]

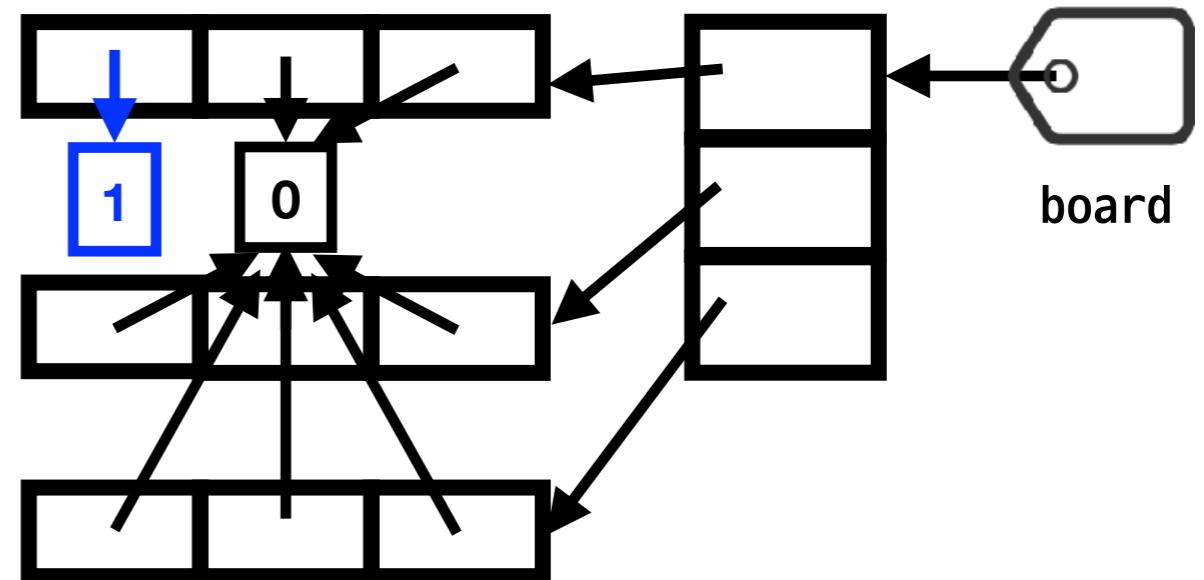
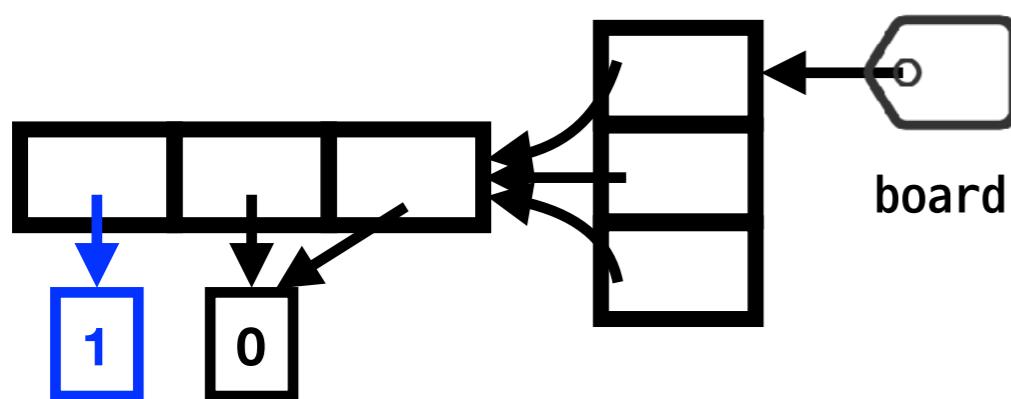


board[0] = [1, 0, 0]
board[1] = [1, 0, 0]
board[2] = [1, 0, 0]

board[0][0] = 1 결과

board = [[0] * cols] * rows

board = [[0] * cols for _ in range(rows)]

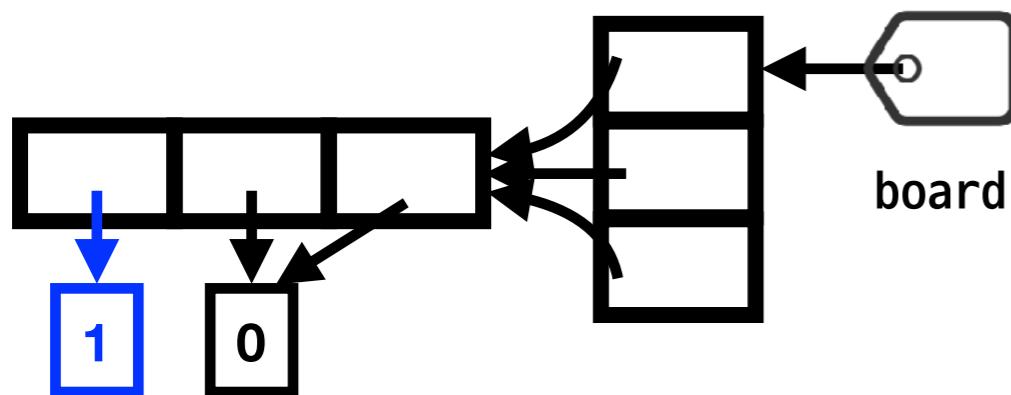


board[0] = [1, 0, 0]
board[1] = [1, 0, 0]
board[2] = [1, 0, 0]

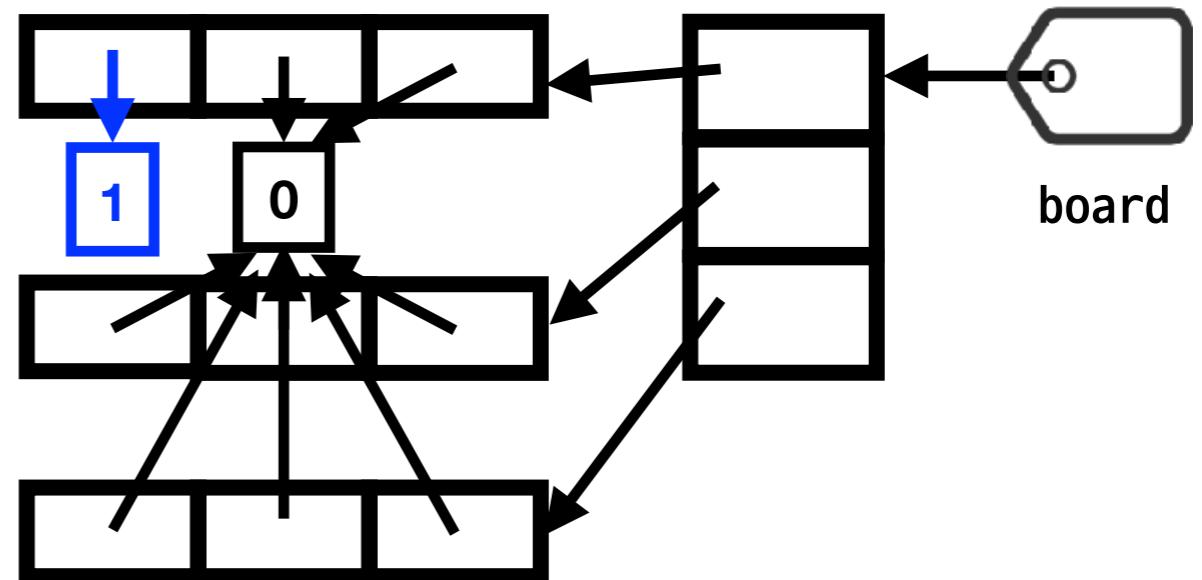
Bad!!

board[0][0] = 1 결과

board = [[0] * cols] * rows



board = [[0] * cols for _ in range(rows)]



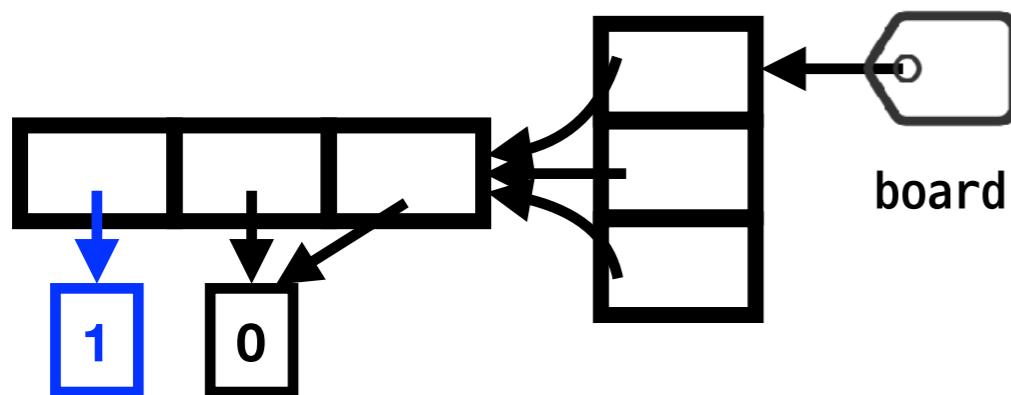
board[0] = [1, 0, 0]
board[1] = [1, 0, 0]
board[2] = [1, 0, 0]

Bad!!

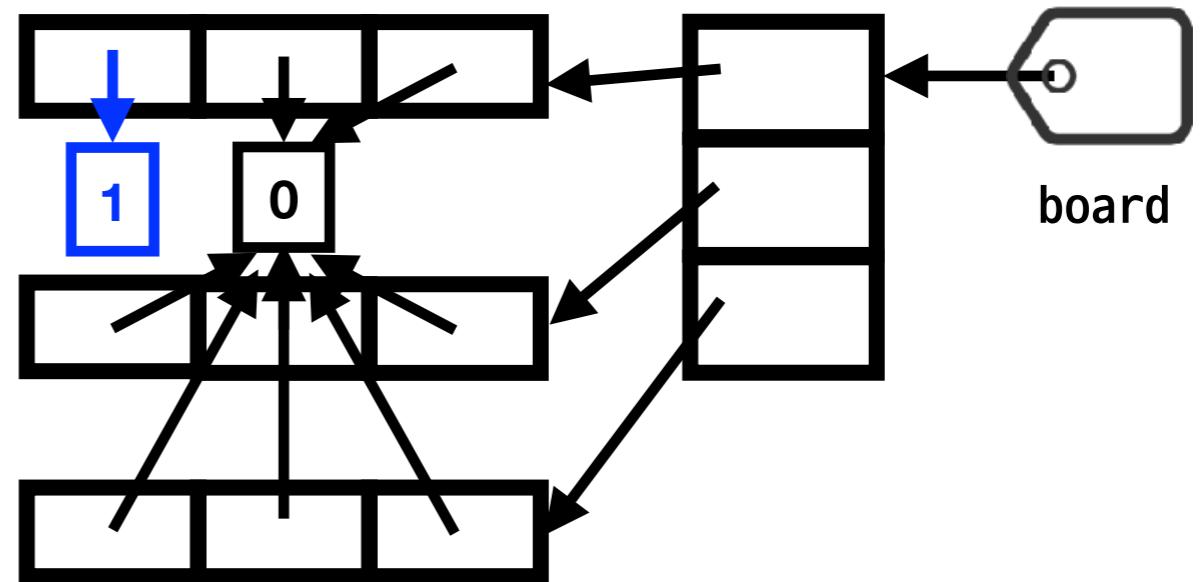
board[0] = [1, 0, 0]
board[1] = [0, 0, 0]
board[2] = [0, 0, 0]

board[0][0] = 1 결과

board = [[0] * cols] * rows



board = [[0] * cols for _ in range(rows)]



board[0] = [1, 0, 0]
board[1] = [1, 0, 0]
board[2] = [1, 0, 0]

Bad!!

board[0] = [1, 0, 0]
board[1] = [0, 0, 0]
board[2] = [0, 0, 0]

Good!!

가변 vs 불변 속성

mutable vs immutable

- 가변속성 vs 불변속성
- 가변속성(mutable)
 - 객체의 아이덴티티를 바꾸지 않고도 객체값을 수정할 수 있음
- 불변속성(immutable)
 - 객체의 아이덴티티를 바꾸지 않고서는 객체를 고칠 수 없음

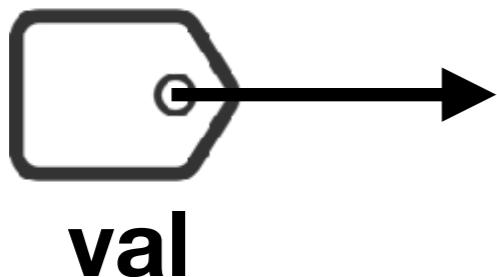
파이썬의 유용한 자료구조

Data structures in python	Mutable or Immutable ?
List	Mutable
Tuple	Immutable
Set	Mutable
Frozen set	Immutable
Dictionary	Mutable

<https://medium.com/@muralielavarthi/data-structures-in-python-9463e6d7cd09>

불변속성(str 형)

```
[>>> val = "immutable val"
[>>> val[0] = 'I'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```



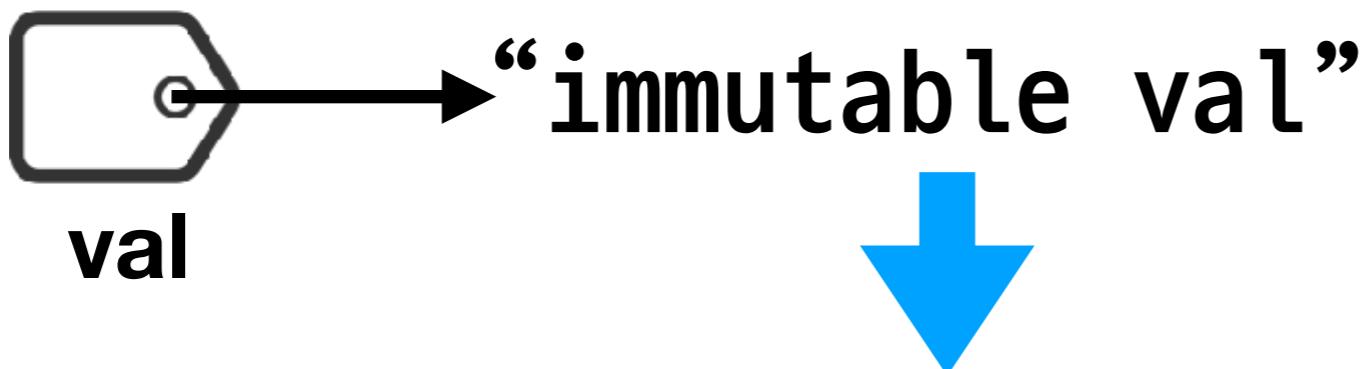
불변속성(str 형)

```
[>>> val = "immutable val"
[>>> val[0] = 'I'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```



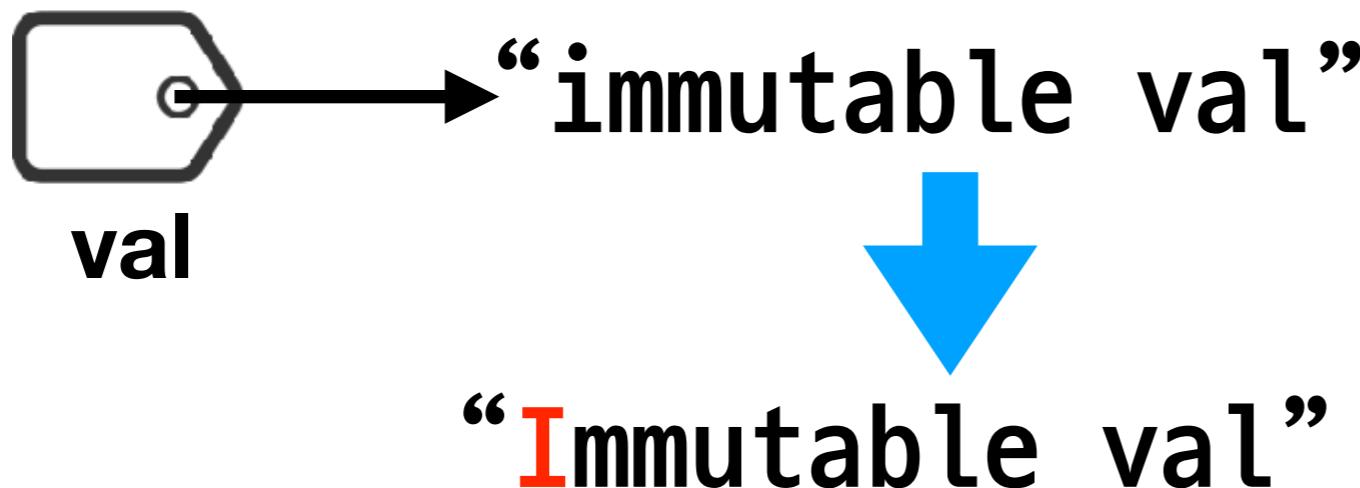
불변속성(str 형)

```
[>>> val = "immutable val"
[>>> val[0] = 'I'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```



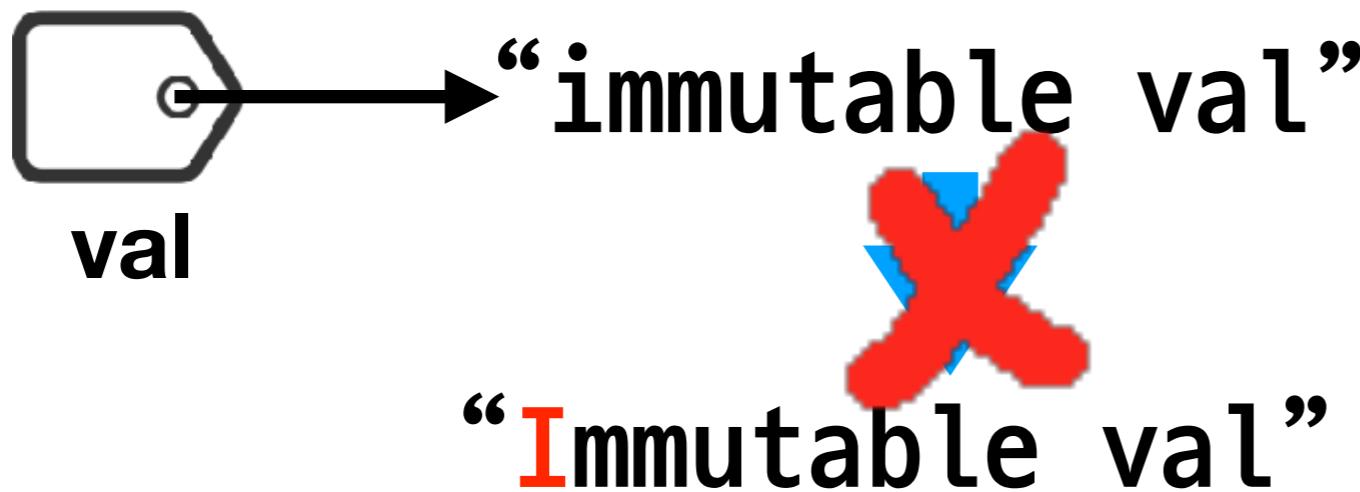
불변속성(str 형)

```
[>>> val = "immutable val"
[>>> val[0] = 'I'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```



불변속성(str 형)

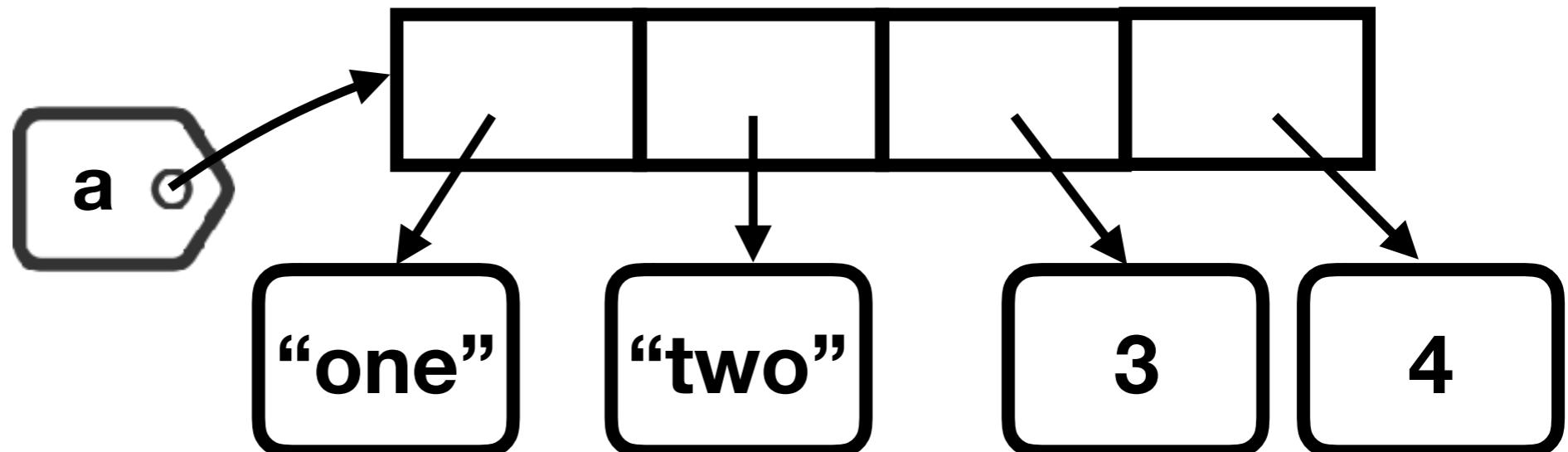
```
[>>> val = "immutable val"
[>>> val[0] = 'I'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```



가변속성(list 형)

리스트 객체 생성

```
a = ["one", "two", 3, 4]
```



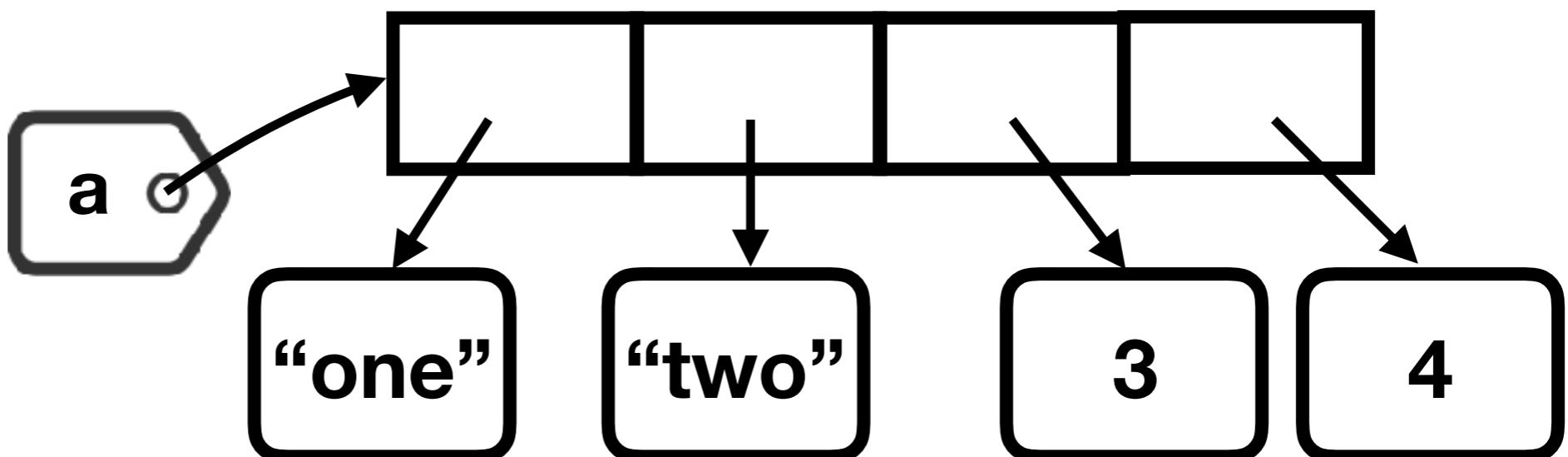
가변속성(list 형)

리스트 객체 생성

a = [“one”, “two”, 3, 4]

리스트 요소의 재할당

a[1] = 2



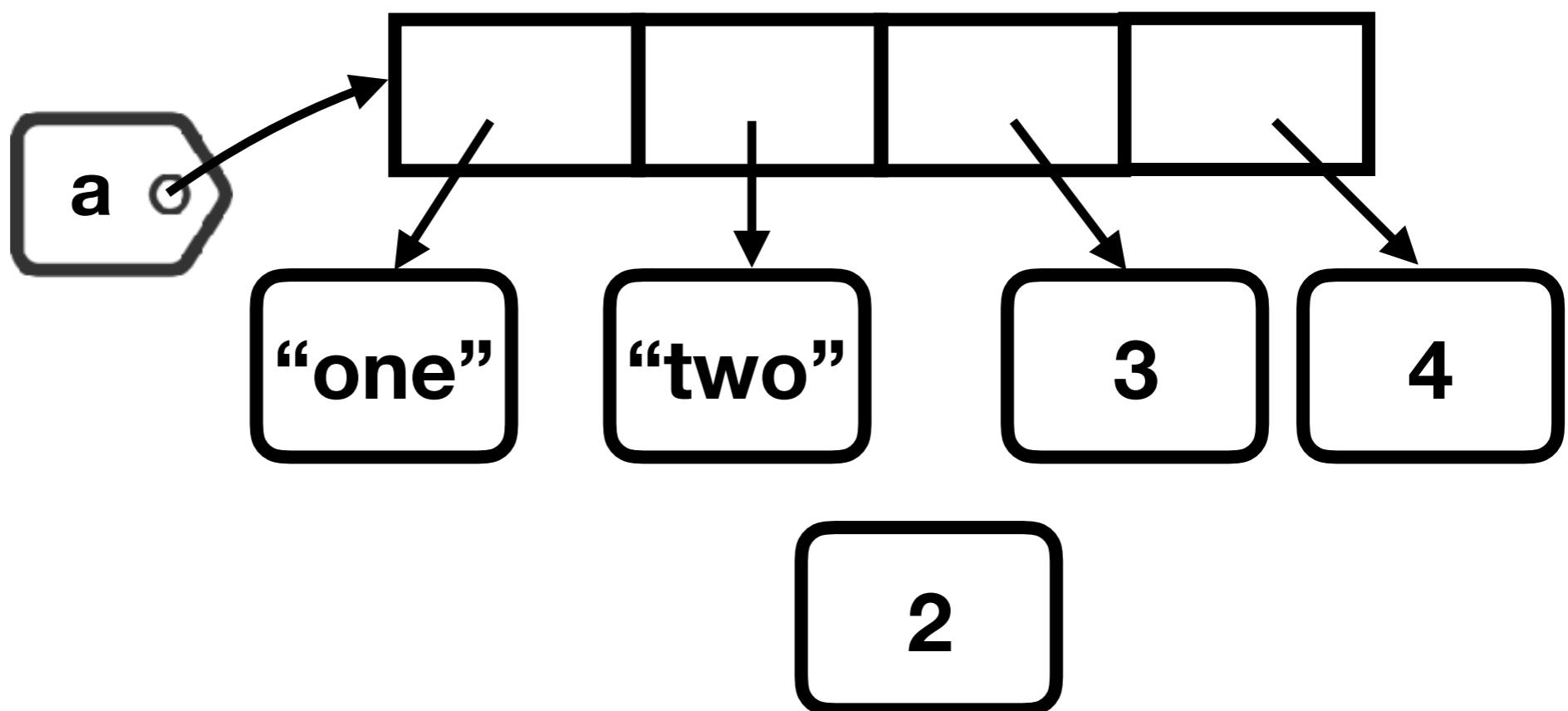
가변속성(list 형)

리스트 객체 생성

a = [“one”, “two”, 3, 4]

리스트 요소의 재할당

a[1] = 2



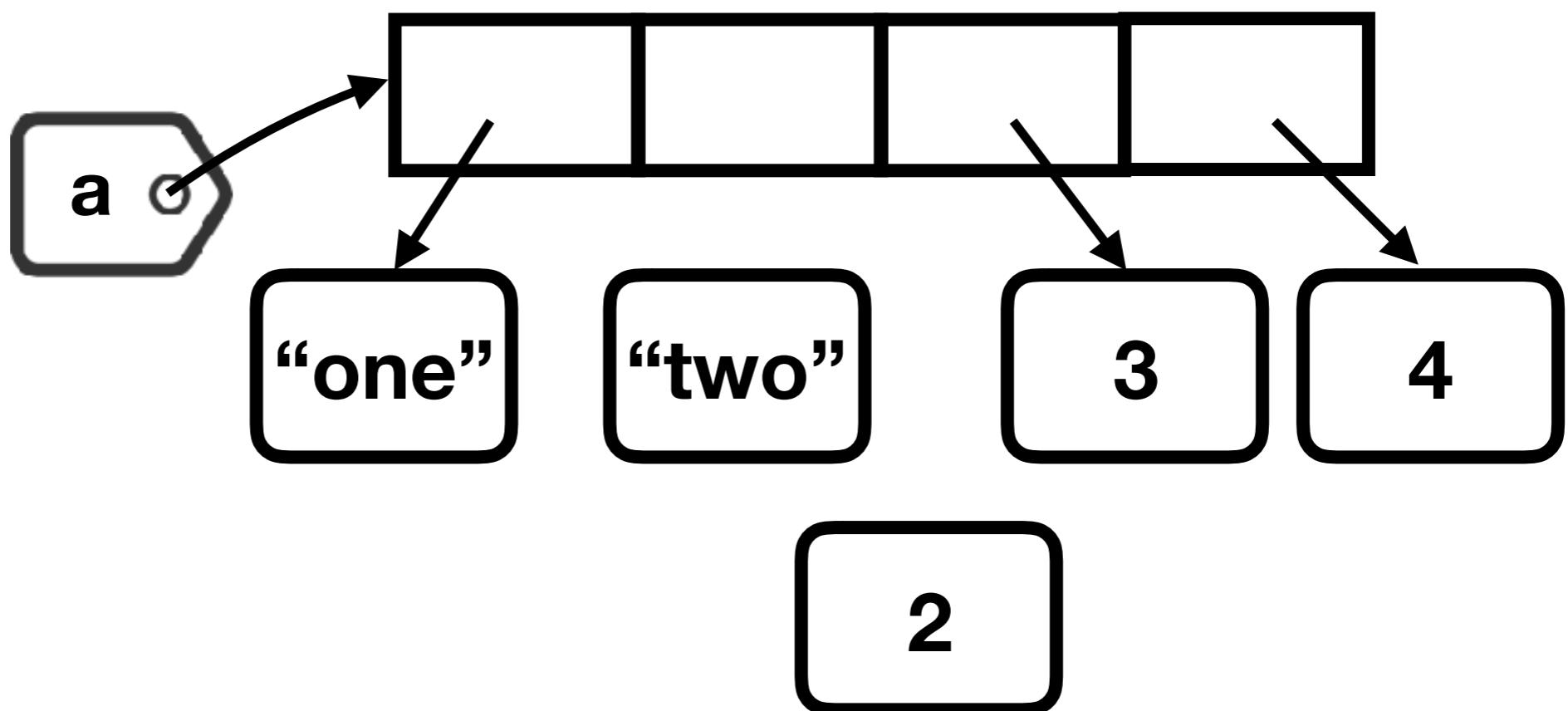
가변속성(list 형)

리스트 객체 생성

a = [“one”, “two”, 3, 4]

리스트 요소의 재할당

a[1] = 2



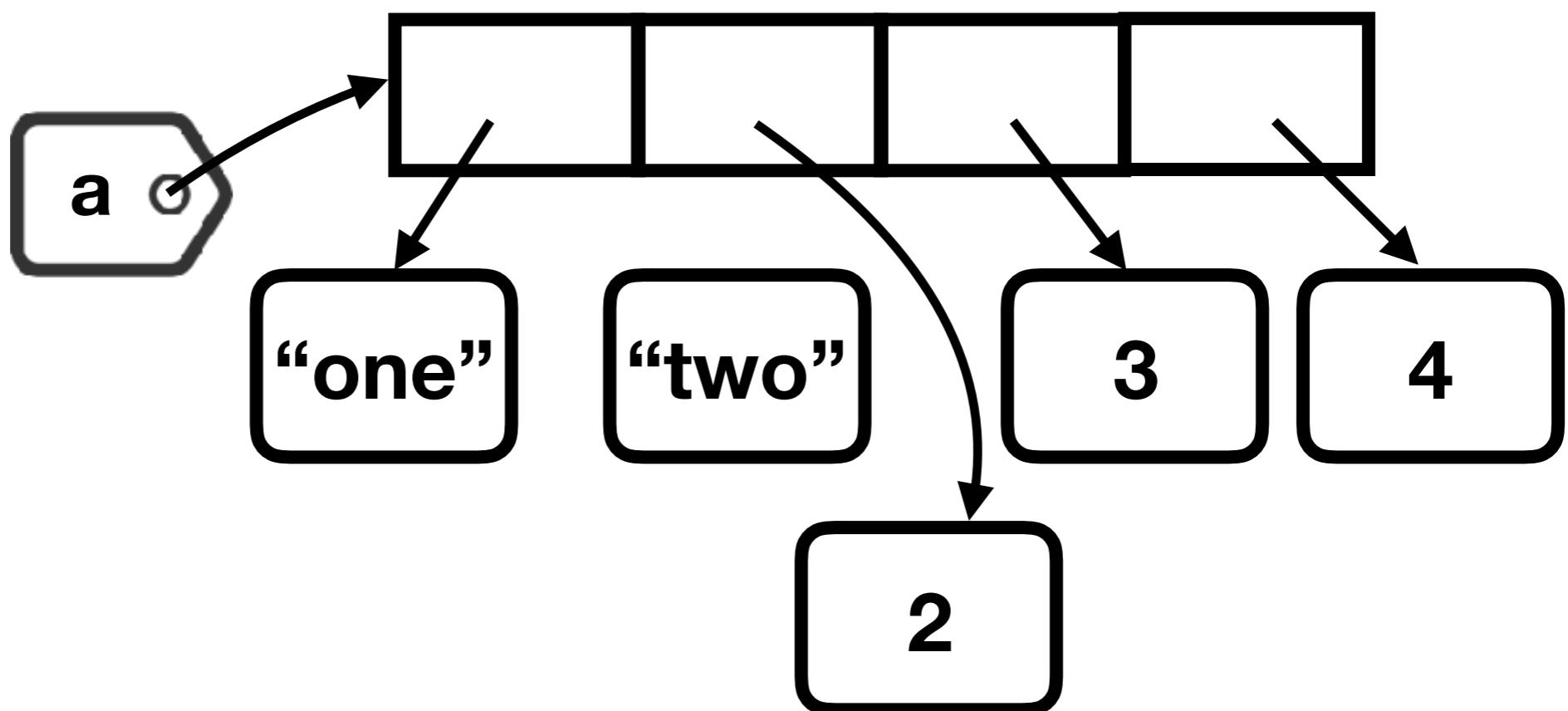
가변속성(list 형)

리스트 객체 생성

a = [“one”, “two”, 3, 4]

리스트 요소의 재할당

a[1] = 2



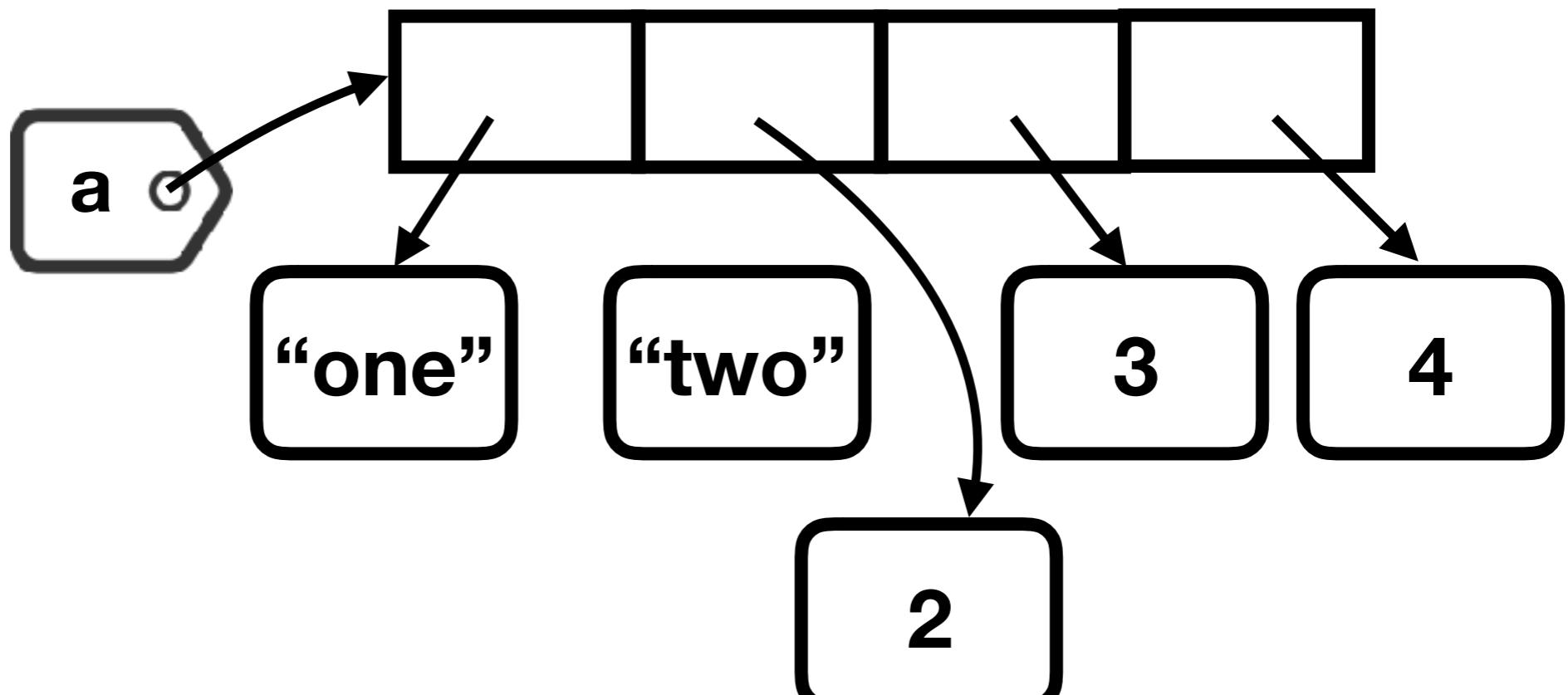
가변속성(list 형)

리스트 객체 생성

```
a = ["one", "two", 3, 4]
```

리스트 요소의 재할당

```
a[1] = 2
```



a 변수가 참조하는 리스트 객체의 내용이 변경가능(가변적)하다
리스트에 새 요소가 들어올때 객체의 id가 바뀔까요? NO

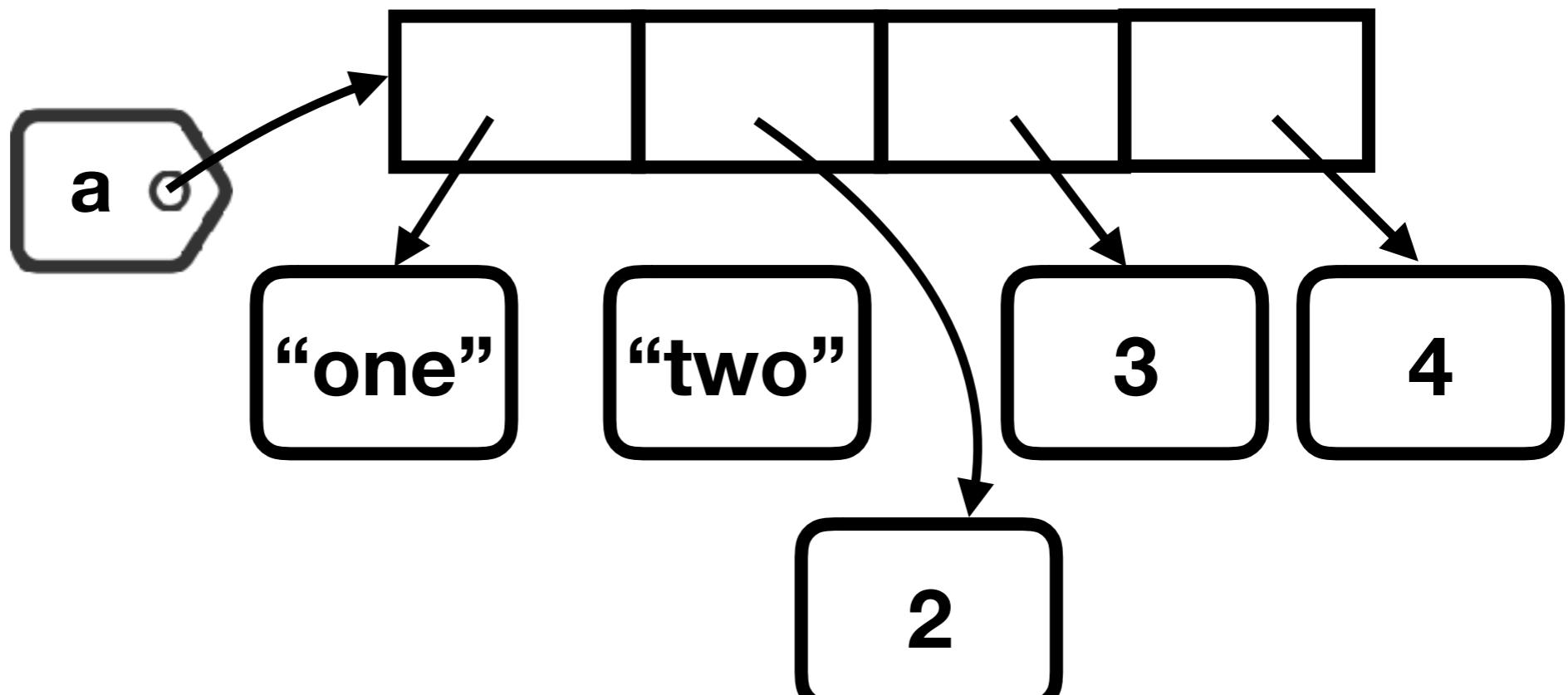
가변속성(list 형)

리스트 객체 생성

```
a = ["one", "two", 3, 4]
```

리스트 요소의 재할당

```
a[1] = 2
```



a 변수가 참조하는 리스트 객체의 내용이 변경가능(가변적)하다
리스트에 새 요소가 들어올때 객체의 id가 바뀔까요? NO

불변(immutable) 속성

```
[>>> val = "im-mutable val"  
[>>> id(val)  
4525124976  
[>>> val = "immutable val"  
[>>> id(val)  
4525125104
```

가변(mutable) 속성

```
[>>> mylist = []  
[>>> id(mylist)  
4524770160  
[>>> mylist.append('Hello')  
[>>> mylist.append('Python')  
[>>> id(mylist)  
4524770160
```

불변(immutable) 속성

```
[>>> val = "im-mutable val"  
[>>> id(val)  
4525124976  
[>>> val = "immutable val"  
[>>> id(val)  
4525125104
```

변수에 다른 객체를 할당하니
아이덴티티 값이 변하더라(??)

가변(mutable) 속성

```
[>>> mylist = []  
[>>> id(mylist)  
4524770160  
[>>> mylist.append('Hello')  
[>>> mylist.append('Python')  
[>>> id(mylist)  
4524770160
```

불변(immutable) 속성

```
[>>> val = "im-mutable val"  
[>>> id(val)  
4525124976  
[>>> val = "immutable val"  
[>>> id(val)  
4525125104
```

변수에 다른 객체를 할당하니
아이덴티티 값이 변하더라(??)

가변(mutable) 속성

```
[>>> mylist = []  
[>>> id(mylist)  
4524770160  
[>>> mylist.append('Hello')  
[>>> mylist.append('Python')  
[>>> id(mylist)  
4524770160
```

불변(immutable) 속성

```
[>>> val = "im-mutable val"  
[>>> id(val)  
4525124976  
[>>> val = "immutable val"  
[>>> id(val)  
4525125104
```

변수에 다른 객체를 할당하니
아이덴티티 값이 변하더라(??)

아이덴티티가 변하는 데 왜 불변 속성이니?

가변(mutable) 속성

```
[>>> mylist = []  
[>>> id(mylist)  
4524770160  
[>>> mylist.append('Hello')  
[>>> mylist.append('Python')  
[>>> id(mylist)  
4524770160
```

불변(immutable) 속성

```
[>>> val = "im-mutable val"  
[>>> id(val)  
4525124976  
[>>> val = "immutable val"  
[>>> id(val)  
4525125104
```

변수에 다른 객체를 할당하니
아이덴티티 값이 변하더라(??)

아이덴티티가 변하는 데 왜 불변 속성이니?

변수가 변할 뿐 객체는 안변하니 불변속성(immutable)

가변(mutable) 속성

```
[>>> mylist = []  
[>>> id(mylist)  
4524770160  
[>>> mylist.append('Hello')  
[>>> mylist.append('Python')  
[>>> id(mylist)  
4524770160
```

불변(immutable) 속성

```
[>>> val = "im-mutable val"  
[>>> id(val)  
4525124976  
[>>> val = "immutable val"  
[>>> id(val)  
4525125104
```

변수에 다른 객체를 할당하니
아이덴티티 값이 변하더라(??)

아이덴티티가 변하는 데 왜 불변 속성이니?

변수가 변할 뿐 객체는 안변하니 불변속성(immutable)

가변(mutable) 속성

```
[>>> mylist = []  
[>>> id(mylist)  
4524770160  
[>>> mylist.append('Hello')  
[>>> mylist.append('Python')  
[>>> id(mylist)  
4524770160
```

mylist 객체의 아이덴티티는 안 변하더라

불변(immutable) 속성

```
[>>> val = "im-mutable val"  
[>>> id(val)  
4525124976  
[>>> val = "immutable val"  
[>>> id(val)  
4525125104
```

변수에 다른 객체를 할당하니
아이덴티티 값이 변하더라(??)

아이덴티티가 변하는 데 왜 불변 속성이니?

변수가 변할 뿐 객체는 안변하니 불변속성(immutable)

가변(mutable) 속성

```
[>>> mylist = []  
[>>> id(mylist)  
4524770160  
[>>> mylist.append('Hello')  
[>>> mylist.append('Python')  
[>>> id(mylist)  
4524770160
```

mylist 객체의 아이덴티티는 안 변하더라

불변(immutable) 속성

```
[>>> val = "im-mutable val"  
[>>> id(val)  
4525124976  
[>>> val = "immutable val"  
[>>> id(val)  
4525125104
```

변수에 다른 객체를 할당하니
아이덴티티 값이 변하더라(??)

아이덴티티가 변하는 데 왜 불변 속성이니?

변수가 변할 뿐 객체는 안변하니 불변속성(immutable)

가변(mutable) 속성

```
[>>> mylist = []  
[>>> id(mylist)  
4524770160  
[>>> mylist.append('Hello')  
[>>> mylist.append('Python')  
[>>> id(mylist)  
4524770160
```

mylist 객체의 아이덴티티는 안 변하더라

```
>>> mylist = mylist + ['!']  
>>> id(mylist)  
4525096704
```

불변(immutable) 속성

```
[>>> val = "im-mutable val"  
[>>> id(val)  
4525124976  
[>>> val = "immutable val"  
[>>> id(val)  
4525125104
```

변수에 다른 객체를 할당하니
아이덴티티 값이 변하더라(??)

아이덴티티가 변하는 데 왜 불변 속성이니?

변수가 변할 뿐 객체는 안변하니 불변속성(immutable)

가변(mutable) 속성

```
[>>> mylist = []  
[>>> id(mylist)  
4524770160  
[>>> mylist.append('Hello')  
[>>> mylist.append('Python')  
[>>> id(mylist)  
4524770160
```

mylist 객체의 아이덴티티는 안 변하더라

```
>>> mylist = mylist + ['!']  
>>> id(mylist)  
4525096704
```

가변객체라도 재바인딩 되면 객체의 아이덴티티는
변한다는 점에 주의!!!

불변(immutable) 속성

```
[>>> val = "im-mutable val"  
[>>> id(val)  
4525124976  
[>>> val = "immutable val"  
[>>> id(val)  
4525125104
```

변수에 다른 객체를 할당하니
아이덴티티 값이 변하더라(??)

아이덴티티가 변하는 데 왜 불변 속성이니?

변수가 변할 뿐 객체는 안변하니 불변속성(immutable)

가변(mutable) 속성

```
[>>> mylist = []  
[>>> id(mylist)  
4524770160  
[>>> mylist.append('Hello')  
[>>> mylist.append('Python')  
[>>> id(mylist)  
4524770160
```

mylist 객체의 아이덴티티는 안 변하더라

```
>>> mylist = mylist + ['!']  
>>> id(mylist)  
4525096704
```

가변객체라도 재바인딩 되면 객체의 아이덴티티는
변한다는 점에 주의!!!

정리

- Life is short - you need Python!
- 향후 3-4년 이내에 가장 널리 사용되는 프로그래밍 언어가 될 수 있음
- 머신러닝을 배울적에 파이썬 공부를 깊이있게 하시기를 권합니다
- 동적 특성과 가변, 불변속성등 다소 생소한 내용에 대한 개념이해도 필요합니다
- 파이썬의 특징을 이해하고, 더욱 더 강력하게 파이썬을 이용하기 바랍니다.

C 언어에 익숙하신 분들께

C 언어에 익숙하신 분들께

- 파이썬은 C 언어와 매우 많은 면에서 다른 언어입니다

C 언어에 익숙하신 분들께

- 파이썬은 C 언어와 매우 많은 면에서 다른 언어입니다
- 열린 마음으로 객체지향 언어의 장점을 받아들이신다면

C 언어에 익숙하신 분들께

- 파이썬은 C 언어와 매우 많은 면에서 다른 언어입니다
- 열린 마음으로 객체지향 언어의 장점을 받아들이신다면
- 매우 즐겁게 파이썬을 배우실 수 있습니다

감사합니다.

Questions??

