

데이터과학을 위한
**파이썬
프로그래밍**



03. 화면 입출력과 리스트

목차

1. 파이썬 프로그래밍 환경
2. 화면 입출력
3. Lab: 화씨온도 변환기
4. 리스트의 이해
5. 리스트의 메모리 관리 방식

01

파이썬 프로그래밍 환경

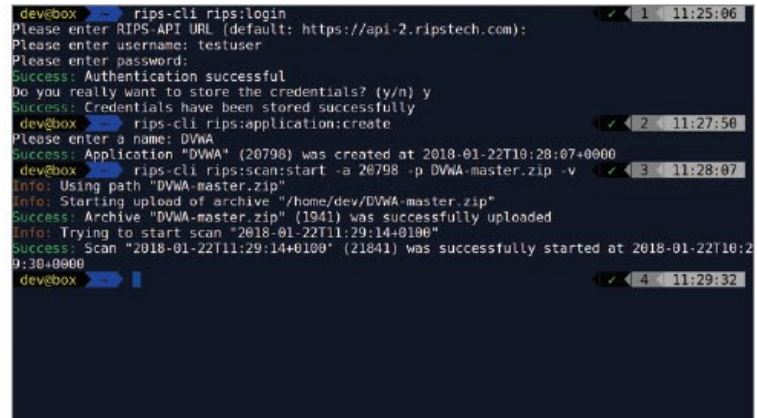
01. 파이썬 프로그래밍 환경

■ 사용자 인터페이스

- 컴퓨터에 명령을 입력할 때 사용하는 환경을 사용자 인터페이스(user interface)라고 한다.



(a) GUI



(b) CLI

[사용자 인터페이스]

01. 파이썬 프로그래밍 환경

■ CLI 환경

- 사용자 인터페이스에는 GUI(Graphical User Interface) 와 CLI(Command Line Interface) 가 있다. 일반적으로 GUI 환경에서는 아이콘을 하나 클릭하면 그에 대한 명령이 실행된다. CLI 는 마우스의 클릭이 아닌 키보드만으로 명령을 입력하는 환경이다.

02

화면 입출력

02. 화면 입출력

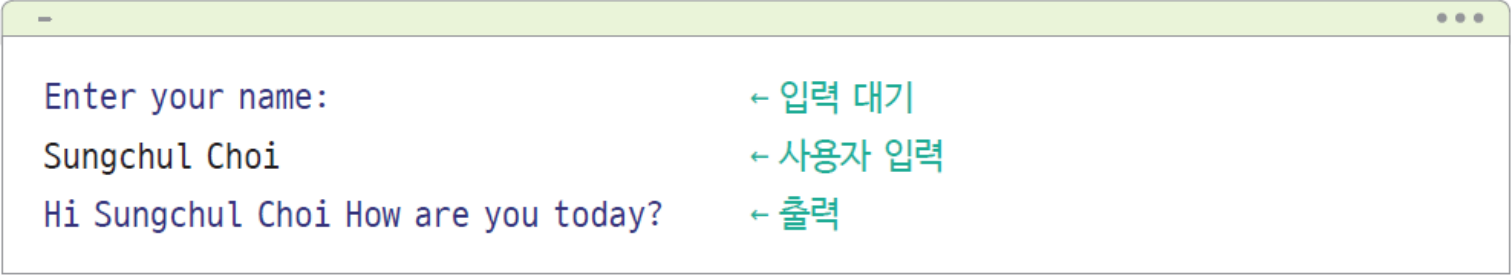
예제: input.py

■ 표준 입력 함수: input() 함수

- **input() 함수:** 표준 입력 함수로, 사용자가 문자열을 콘솔 창에 입력할 수 있게 해 준다.

코드 3-1 input.py

```
1 print("Enter your name:")
2 somebody = input()           # 콘솔 창에서 입력한 값을 somebody에 저장
3 print("Hi", somebody, "How are you today?")
```



```
Enter your name:                ← 입력 대기
Sungchul Choi                  ← 사용자 입력
Hi Sungchul Choi How are you today? ← 출력
```

```
D:\workspace\Ch03>python input.py
Enter your name:
Sungchul Choi
Hi Sungchul Choi How are you today?
```

[input.py를 cmd 창에서 실행]

02. 화면 입출력

예제: print.py

■ 표준 출력 함수: print() 함수

- **print() 함수:** 표준 출력 함수로, 결과를 화면에 출력하는 함수이다.

```
>>> print("Hello World!", "Hello Again!!!")  
Hello World! Hello Again!!!
```

콤마 사용
실행 시 두 문장이 연결되어 출력

코드 3-2 print.py

```
1 temperature = float(input("온도를 입력하세요: "))  
2 print(temperature)
```

입력 시 바로 형 변환

```
온도를 입력하세요: 103  
103.0
```

← 입력 대기 및 사용자 입력
← 출력

03

Lab: 화씨온도 변환기

03. Lab: 화씨온도 변환기

예제: `fahrenheit.py`

■ 실습 내용

- 이번 Lab에서는 앞에서 배운 `input()` 함수, `print()` 함수, 간단한 사칙연산을 이용하여 화씨 온도 변환기(Fahrenheit temperature converter) 프로그램을 만든다.
- 섭씨온도와 화씨온도의 변환 공식은 다음과 같다.

$$\text{화씨온도(°F)} = (\text{섭씨온도(°C)} * 1.8) + 32$$

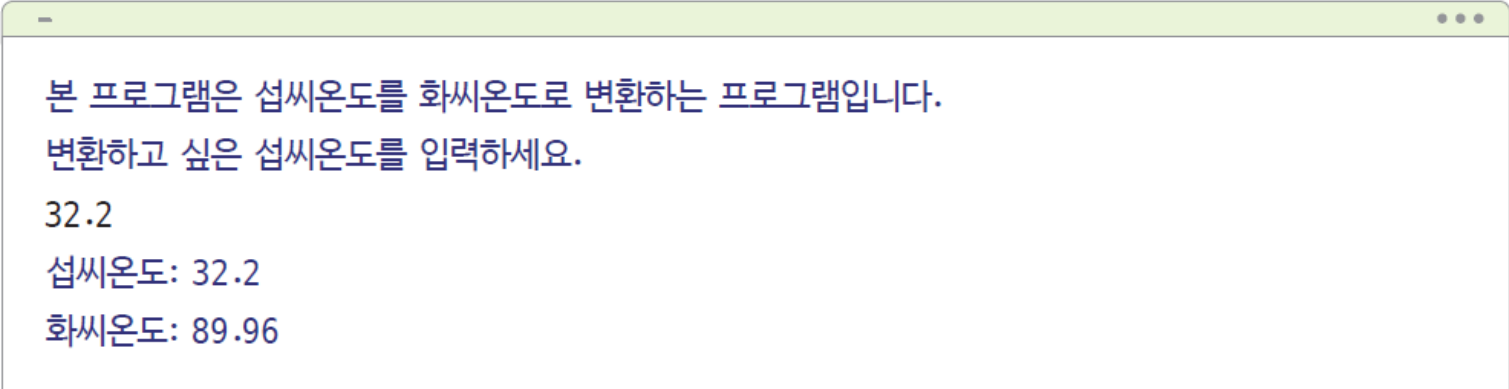
03. Lab: 화씨온도 변환기

예제: fahrenheit.py

■ 실행 결과

- 화씨온도 변환기 프로그램의 실행 결과는 다음과 같다.

실습 0: fahrenheit.py



```
-  
본 프로그램은 섭씨온도를 화씨온도로 변환하는 프로그램입니다.  
변환하고 싶은 섭씨온도를 입력하세요.  
32.2  
섭씨온도: 32.2  
화씨온도: 89.96
```

실습 1: fahrenheit.py

```
본 프로그램은 섭씨온도를 화씨온도로 변환하는 프로그램입니다.  
변환하고 싶은 섭씨온도: 30  
섭씨온도: 30  
화씨온도: 86.0
```

03. Lab: 화씨온도 변환기

예제: fahrenheit.py

■ 문제 해결

- 화씨온도 변환기 프로그램의 결과 코드는 [코드 3-3]과 같다.

코드 3-3 fahrenheit.py

```
1 print("본 프로그램은 섭씨온도를 화씨온도로 변환하는 프로그램입니다.")
2 print("변환하고 싶은 섭씨온도를 입력하세요.")
3
4 celsius = input()
5 fahrenheit = (float(celsius) * 1.8 ) + 32
6
7 print("섭씨온도:", celsius)
8 print("화씨온도:", fahrenheit)
```

$$\text{화씨온도(°F)} = (\text{섭씨온도(°C)} * 1.8) + 32$$

04

리스트의 이해

04. 리스트의 이해

■ 리스트의 개념

- **리스트(list):** 하나의 변수에 여러 값을 할당하는 자료형이다.
- 파이썬에서는 리스트처럼 여러 데이터를 하나의 변수에 할당하는 기법을 시퀀스 자료형이라고 한다. 시퀀스 자료형은 여러 자료를 순서대로 넣는다는 뜻이다.
- 리스트는 하나의 자료형으로만 저장하지 않고, 정수형이나 실수형 같은 다양한 자료형을 포함할 수 있다.

```
colors = ['red', 'blue', 'green']
```

colors →

'red'	'blue'	'green'
-------	--------	---------

[리스트의 예]

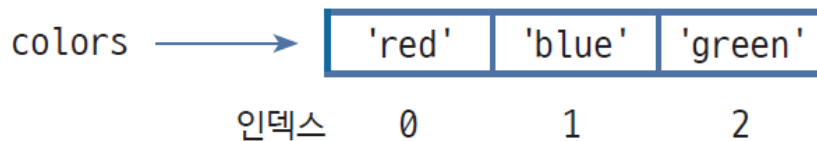
■ 인덱싱과 슬라이싱 : 인덱싱(indexing)

- 인덱싱: 리스트에 있는 값에 접근하기 위해, 이 값의 상대적인 주소를 사용하는 것이다.

코드 3-4 indexing.py

```
1 colors = ['red', 'blue', 'green']
2 print(colors[0])
3 print(colors[2])
4 print(len(colors))
```

```
red
green
3
```



[리스트의 인덱싱]

04. 리스트의 이해

여기서 잠깐! 리스트의 주소값은 왜 0부터 시작하는가?

- 프로그래밍 언어 대부분에서 배열(array) 계열 변수의 주소값은 0부터 시작한다. 여러 가지 이유가 있지만, 일단 1부터 시작하는 것보다 0부터 시작하면 이진수 관점에서 메모리를 절약할 수 있다는 장점이 있다. 또한, 1보다는 0부터 시작하는 것이 진수에서 00부터 사용할 수 있는 장점도 있다. 지금은 큰 문제가 안 되지만, 초기 컴퓨터들은 메모리 절약이 매우 큰 이슈였기 때문에 이 점은 중요했다. 비주얼 베이직이나 매트랩 같은 언어에서는 1부터 인덱싱을 하기도 하니 알아두면 좋다.

■ 인덱싱과 슬라이싱 : 슬라이싱(slicing)

- 슬라이싱: 리스트의 인덱스를 사용하여 전체 리스트에서 일부를 잘라내어 반환한다.

```
>>> cities = ['서울', '부산', '인천', '대구', '대전', '광주', '울산', '수원']
```

값	['서울', '부산', '인천', '대구', '대전', '광주', '울산', '수원']							
인덱스	0	1	2	3	4	5	6	7

[cities 변수의 리스트]

■ 인덱싱과 슬라이싱 : 슬라이싱(slicing)

- 슬라이싱의 기본 문법

변수명[시작 인덱스:마지막 인덱스] [시작:끝+1]

```
>>> cities = ['서울', '부산', '인천', '대구', '대전', '광주', '울산', '수원']
>>> cities[0:6]
['서울', '부산', '인천', '대구', '대전', '광주']
```

- 파이썬의 리스트에서 '마지막 인덱스 - 1'까지만 출력된다. 만약 한 번 이상 리스트 변수를 사용하면 마지막 인덱스가 다음 리스트의 시작 인덱스가 되어 코드를 작성할 때 조금 더 쉽게 이해할 수 있다는 장점이 있다.

```
>>> cities[0:5]
['서울', '부산', '인천', '대구', '대전']
>>> cities[5:]
['광주', '울산', '수원']
```

■ 인덱싱과 슬라이싱 : 리버스 인덱스(reverse index)

- 리스트에는 인덱스를 마지막 값부터 시작하는 리버스 인덱스 기능이 있다.

값	['서울', '부산', '인천', '대구', '대전', '광주', '울산', '수원']							
인덱스	-8	-7	-6	-5	-4	-3	-2	-1

[cities 변수의 리버스 인덱스]

- 일반적으로 시작 인덱스가 비어 있으면 처음부터, 마지막 인덱스가 비어 있으면 마지막까지라는 의미로 사용된다. 즉, cities[-8:]은 인덱스가 -8인 '서울'부터 '수원'까지 출력하라는 뜻이다.

```
>>> cities = ['서울', '부산', '인천', '대구', '대전', '광주', '울산', '수원']
>>> print(cities[-8:])
['서울', '부산', '인천', '대구', '대전', '광주', '울산', '수원']
```

■ 인덱싱과 슬라이싱 : 인덱스 범위를 넘어가는 슬라이싱

- 인덱스를 따로 넣지 않고 `print(cities[:])`과 같이 콜론(:)을 넣으면 `cities` 변수의 모든 값을 다 반환한다.
- 슬라이싱에서는 인덱스를 넘어서거나 입력하지 않더라도 자동으로 시작 인덱스와 마지막 인덱스로 지정된다

```
>>> cities = ['서울', '부산', '인천', '대구', '대전', '광주', '울산', '수원']
>>> print(cities[:])           # cities 변수의 처음부터 끝까지
['서울', '부산', '인천', '대구', '대전', '광주', '울산', '수원']
>>> print(cities[-50:50])      # 범위를 넘어갈 경우 자동으로 최대 범위를 지정
['서울', '부산', '인천', '대구', '대전', '광주', '울산', '수원']
```

■ 인덱싱과 슬라이싱 : 증가값(step)

- 슬라이싱에서는 시작 인덱스와 마지막 인덱스 외에 마지막 자리에 증분(increment)을 넣을 수 있다.
- 증분이 음수이면 감소, 양수이면 증가를 의미한다.

변수명[시작 인덱스:마지막 인덱스:증가값]

범위 지정 방법=>[시작:끝+1:증분]

```
>>> cities = ['서울', '부산', '인천', '대구', '대전', '광주', '울산', '수원']
>>> cities[::2]                                # 2칸 단위로
['서울', '인천', '대전', '울산']
>>> cities[::-1]                              # 역으로 슬라이싱
['수원', '울산', '광주', '대전', '대구', '인천', '부산', '서울']
```

■ 리스트의 연산

- 덧셈 연산 : 덧셈 연산을 하더라도 따로 어딘가 변수에 할당해 주지 않으면 기존 변수는 변화가 없다

```
>>> color1 = ['red', 'blue', 'green']
>>> color2 = ['orange', 'black', 'white']
>>> print(color1 + color2)           # 두 리스트 합치기
['red', 'blue', 'green', 'orange', 'black', 'white']
>>> len(color1)                     # 리스트 길이
3
>>> total_color = color1 + color2
>>>
```

```
>>> total_color = color1 + color2
>>> total_color
['red', 'blue', 'green', 'orange', 'black', 'white']
```

■ 리스트의 연산

- **곱셈 연산** : 리스트의 곱셈 연산은 기준 리스트에 n을 곱했을 때, 같은 리스트를 n배만큼 늘려 준다.

```
>>> color1 * 2                                # color1 리스트 2회 반복  
['red', 'blue', 'green', 'red', 'blue', 'green']
```

- **in 연산** : 포함 여부를 확인하는 연산으로, 하나의 값이 해당 리스트에 들어 있는지 확인할 수 있다.

```
>>> 'blue' in color2                            # color2 변수에서 문자열 'blue'의 존재 여부 반환  
False
```

■ 리스트 추가 및 삭제

- **append() 함수** : 새로운 값을 기존 리스트의 맨 끝에 추가

```
>>> color = ['red', 'blue', 'green']
>>> color.append('white')           # 리스트에 'white' 추가
>>> color
['red', 'blue', 'green', 'white']
```

- **extend() 함수** : 새로운 리스트를 기존 리스트에 추가

```
>>> color = ['red', 'blue', 'green']
>>>
>>> color.extend(['black', 'purple']) # 리스트에 새로운 리스트 추가
>>> color
['red', 'blue', 'green', 'black', 'purple']
```


■ 리스트 추가 및 삭제

- **insert() 함수** : 기존 리스트의 i번째 인덱스에 새로운 값을 추가, i번째 인덱스를 기준으로 뒤쪽의 인덱스가 하나씩 밀림.

```
>>> color = ['red', 'blue', 'green']
>>>
>>> color.insert(0, 'orange')
>>> color
['orange', 'red', 'blue', 'green']
```

■ 리스트 추가 및 삭제

- **remove() 함수** : 리스트 내의 특정 값을 삭제.

```
>>> color
['orange', 'red', 'blue', 'green']
>>>
>>> color.remove('red')
>>> color
['orange', 'blue', 'green']
```

■ 리스트 추가 및 삭제

- 인덱스의 재할당 : 인덱스에 새로운 값을 할당한다.
- 인덱스 삭제 : del 함수를 사용한다.

```
>>> color = ['red', 'blue', 'green']
>>> color[0] = 'orange'
>>> color
['orange', 'blue', 'green']
>>> del color[0]
>>> color
['blue', 'green']
```

04. 리스트의 이해

예제: list.py 실습 1

■ 리스트 추가 및 삭제

함수	기능	용례
append()	새로운 값을 기존 리스트의 맨 끝에 추가	color.append('white')
extend()	새로운 리스트를 기존 리스트에 추가(덧셈 연산과 같은 효과)	color.extend(['black','purple'])
insert()	기존 리스트의 i번째 인덱스에 새로운 값을 추가. i번째 인덱스를 기준으로 뒤쪽의 인덱스는 하나씩 밀림	color.insert(0, 'orange')
remove()	리스트 내의 특정 값을 삭제	color.remove('white')
del	특정 인덱스값을 삭제	del color[0]

[리스트 추가 및 삭제 함수]

■ 패킹과 언패킹

- 패킹(packing): 한 변수에 여러 개의 데이터를 할당하는 것.
- 언패킹(unpacking): 한 변수의 데이터를 각각의 변수로 반환하는 것.

```
>>> t = [1, 2, 3]
```

```
# 1, 2, 3을 변수 t에 패킹
```

```
>>> a, b, c = t
```

```
# t에 있는 값 1, 2, 3을 변수 a, b, c에 언패킹
```

```
>>> print(t, a, b, c)
```

```
[1, 2, 3] 1 2 3
```

■ 패킹과 언패킹

- 다음 코드처럼 리스트에 값이 3개인데, 5개로 언패킹을 시도한다면 어떤 결과가 나올까?
다음 코드에서 보는 것처럼 언패킹 시 할당받는 변수의 개수가 적거나 많으면 모두 에러가 발생한다.

```
>>> t = [1, 2, 3]
>>> a, b, c, d, e = t
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: not enough values to unpack (expected 5, got 3)
>>> a, b = t
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: too many values to unpack (expected 2)
```

■ 이차원 리스트

- 리스트를 효율적으로 활용하기 위해 여러 개의 리스트를 하나의 변수에 할당하는 이차원 리스트를 사용할 수 있다.
- 이차원 리스트는 표의 칸에 값을 채웠을 때 생기는 값들의 집합이다.

학생	A	B	C	D	E
국어 점수	49	79	20	100	80
수학 점수	43	59	85	30	90
영어 점수	49	79	48	60	100

[이차원 리스트를 설명하기 위한 예]

■ 이차원 리스트

- 이차원 리스트를 하나의 변수로 표현하기 위해서는 다음과 같이 코드를 작성하면 된다
- 이차원 리스트에 인덱싱하여 값에 접근하기 위해서는 다음 코드와 같이 **대괄호 2개**를 사용한다.

```
>>> kor_score = [49, 79, 20, 100, 80]
>>> math_score = [43, 59, 85, 30, 90]
>>> eng_score = [49, 79, 48, 60, 100]
>>> midterm_score = [kor_score, math_score, eng_score]
>>> midterm_score
[[49, 79, 20, 100, 80], [43, 59, 85, 30, 90], [49, 79, 48, 60, 100]]
```

다차원 list는 numpy로 배열하는 것이 유리할 듯....
2차원 이상은 list로 관리하는데 한계가 있어 보인다...

■ 이차원 리스트

- 이차원 리스트를 행렬로 본다면 [0]은 행, [2]는 열을 뜻한다. 즉, 생성된 이차원 리스트에서 [0]은 kor_score, [2]는 C를 의미하여 실행 결과 20을 화면에 출력한다.

```
>>> print(midterm_score[0][2])  
20
```

미션 :

다음과 같이 그 원소 값이 정의된 2차원, 3x4 리스트 a를 정의하시오.

```
# 9 0 -1 -2  
# 1 2 3 4  
# 5 6 7 8
```

05

리스트의 메모리 관리 방식

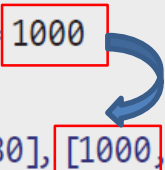
05. 리스트의 메모리 관리 방식

예제: list_memory.py

■ 리스트의 메모리 저장

- 다음 코드에서 가장 핵심 코드는 `math_score[0] = 1000`이다. 분명히 `math_score`의 값을 변경하였는데 `midterm_score` 두 번째 행의 첫 번째 값이 변경되었다. 이는 파이썬 리스트가 값을 저장하는 방식 때문에 발생하는 현상이다.

```
>>> kor_score = [49, 79, 20, 100, 80]
>>> math_score = [43, 59, 85, 30, 90]
>>> eng_score = [49, 79, 48, 60, 100]
>>>
>>> midterm_score = [kor_score, math_score, eng_score]
>>> midterm_score
[[49, 79, 20, 100, 80], [43, 59, 85, 30, 90], [49, 79, 48, 60, 100]]
>>>
>>> math_score[0] = 1000
>>> midterm_score
[[49, 79, 20, 100, 80], [1000, 59, 85, 30, 90], [49, 79, 48, 60, 100]]
```

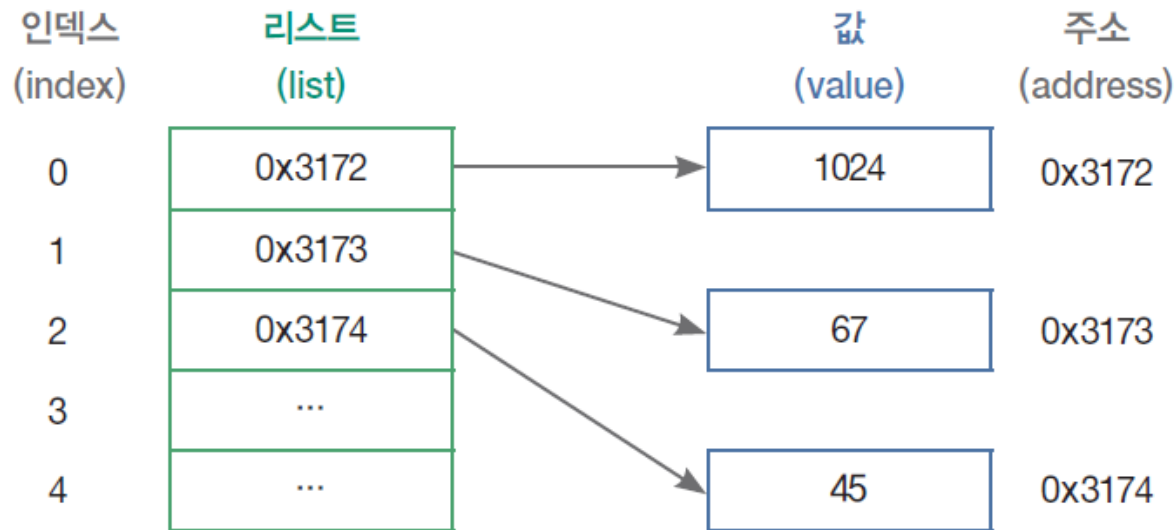


05. 리스트의 메모리 관리 방식

예제: list_memory.py

■ 리스트의 메모리 저장

- 파이썬은 리스트를 저장할 때 값 자체가 아니라, 값이 위치한 메모리 주소(reference)를 저장한다.



[리스트의 메모리 저장]

■ 리스트의 메모리 저장

- ==은 값을 비교하는 연산이고, is는 메모리의 주소를 비교하는 연산이다.
- 아래 코드에서 a와 b는 값은 같지만, 메모리의 저장 주소는 다른 것이다.

```
>>> a = 300
```

```
>>> b = 300
```

```
>>> a is b
```

```
False ← 현재 버전의 파이썬에서는 True로 관측된다.
```

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

```
True
```

05. 리스트의 메모리 관리 방식

예제: list_memory.py

■ 리스트의 메모리 저장

- 이전 코드와 다르게 is와 == 연산자는 모두 True를 반환한다. 그렇다면 a와 b의 메모리 주소는 같은 것일까? 이것은 파이썬의 정수형 저장 방식의 특성 때문이다. 파이썬은 인터프리터가 구동될 때, -5부터 256까지의 정수값을 특정 메모리 주소에 저장한다(??). 그리고 해당 숫자를 할당하려고 하면 해당 변수는 그 숫자가 가진 메모리 주소로 연결한다.

```
>>> a = 1
>>> b = 1
>>> a is b
True
>>> a == b
True
```

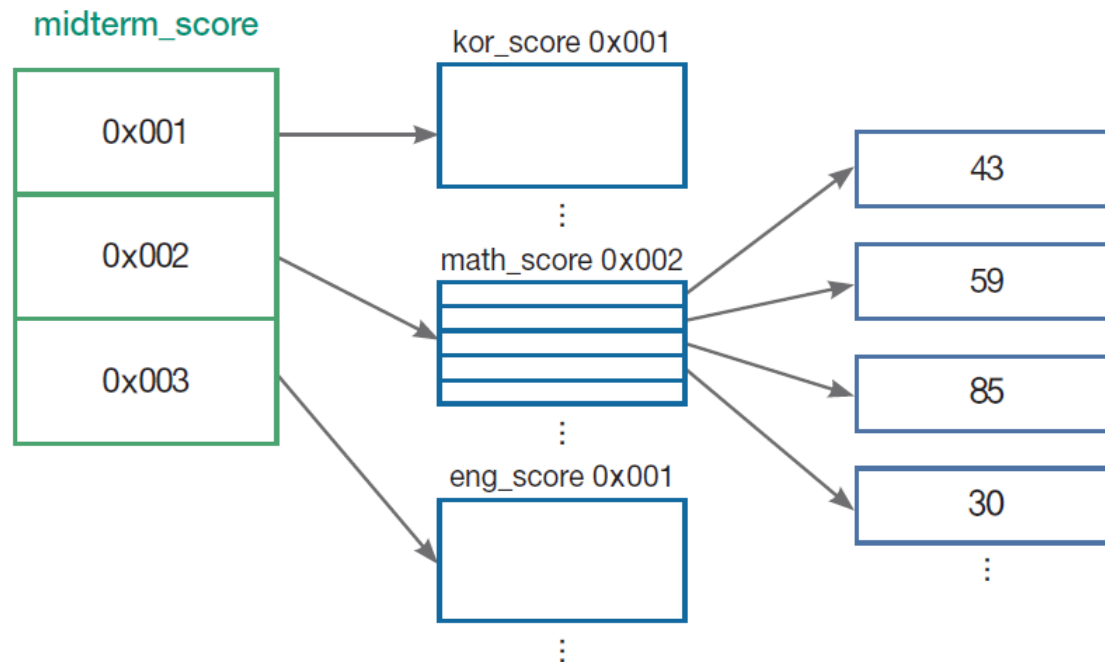
해당 자료를 찾을 수 없음.
실험에 의하면,
현재에는 값에 상관 없이 같은 값을 변수 2개에 배정하면
두 변수는 처음에는 같은 주소의 객체로 할당된다.
예를 들어 두 변수를 같이 300000을 배정해도 같은 주소
가 관찰되었다.

05. 리스트의 메모리 관리 방식

예제: list_memory.py

■ 리스트의 메모리 저장

- 리스트는 기본적으로 값을 연속으로 저장하는 것이 아니라, 값이 있는 주소를 저장하는 방식이다.



[리스트의 메모리 저장 연결 관계]

05. 리스트의 메모리 관리 방식

■ 메모리 저장 구조로 인한 리스트의 특징

- 다양한 형태의 변수가 하나의 리스트에 들어갈 수 있다.

```
>>> a = ["color", 1, 0.2]
```

- 기존 변수들과 함께 리스트 안에 다른 리스트를 넣을 수 있다. 흔히 이를 중첩 리스트라고 한다. 이러한 특징은 파이썬의 리스트가 값이 아닌 메모리의 주소를 저장해 메모리에 새로운 값을 할당하는 데 있어 매우 높은 자유도를 보장하므로 가능하다.

```
>>> a = ["color", 1, 0.2]
>>> color = ['yellow', 'blue', 'green', 'black', 'purple']
>>> a[0] = color                                # 리스트 안에 리스트도 입력 가능
>>> print(a)
[['yellow', 'blue', 'green', 'black', 'purple'], 1, 0.2]
```

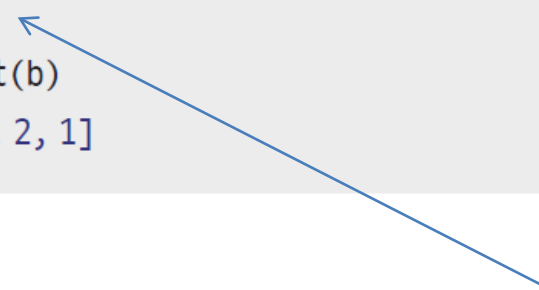

05. 리스트의 메모리 관리 방식

예제: list_memory.py

■ 메모리 저장 구조로 인한 리스트의 특징

- 리스트의 저장 방식
 - b와 a 변수를 각각 다른 값으로 선언한 후, b에 a를 할당하였다. 그리고 b를 출력하면, a 변수와 같은 값이 화면에 출력된다.

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



b는 같은 자료를 복사해서 생성하지 않고, a가 가리키는 지점과 같은 데이터를 가리키는 것으로 대신한다.
따라서 a가 수정되면 b도 같이 수정되는 것이다.

■ 메모리 저장 구조로 인한 리스트의 특징

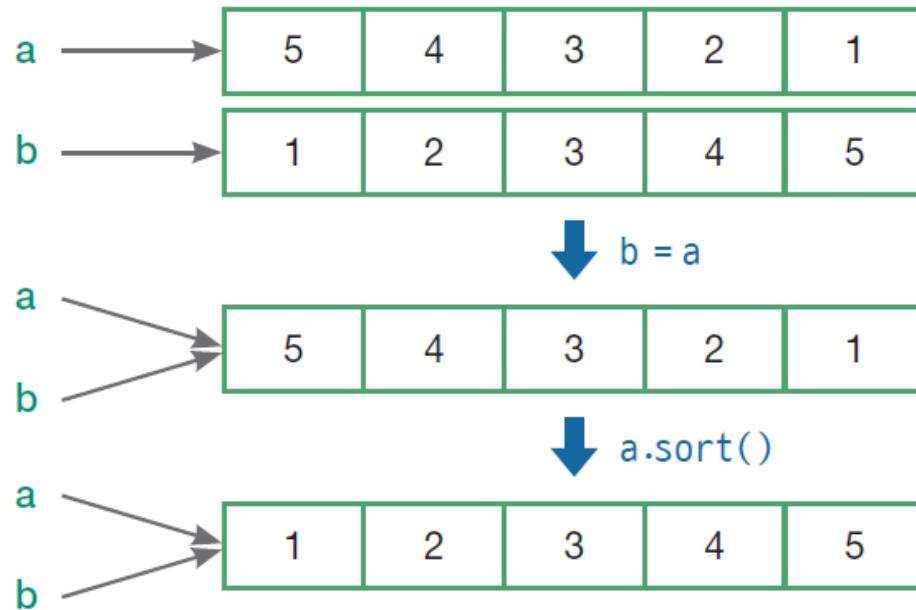
- 리스트의 저장 방식
 - a만 정렬하고 b를 출력했을 때 b도 정렬되었다. 두 변수가 같은 메모리 주소와 연결되어 있으므로, 하나의 변수값만 바뀌더라도 둘 다 영향을 받는다.

```
>>> a.sort()
>>> print(b)
[1, 2, 3, 4, 5]
```

05. 리스트의 메모리 관리 방식

■ 메모리 저장 구조로 인한 리스트의 특징

- 리스트의 저장 방식



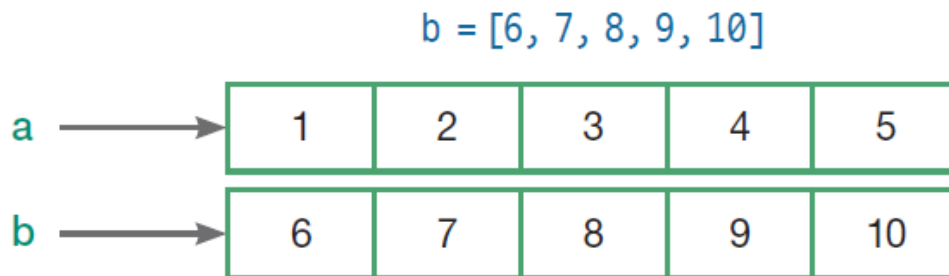
[`a`를 정렬했는데 `b`도 정렬되는 이유]

05. 리스트의 메모리 관리 방식

■ 메모리 저장 구조로 인한 리스트의 특징

- 리스트의 저장 방식
 - b에 새로운 값을 할당하면 b는 이제 새로운 메모리 주소에 새로운 값을 할당할 수 있는 것이다.

```
>>> b = [6, 7, 8, 9, 10]
>>> print(a, b)
[1, 2, 3, 4, 5] [6, 7, 8, 9, 10]
```



[b에 새로운 값을 할당하는 경우]

Thank You !