

데이터 과학을 위한 파이썬 프로그래밍

2판



Chapter 03

화면 입출력과 리스트



목차

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

학습목표

- 사용자 인터페이스를 학습하고, CLI 환경에 대해 이해한다.
- 표준 입력 함수인 `input()` 함수와 표준 출력 함수인 `print()` 함수에 대해 알아본다.
- 리스트의 필요성과 개념에 대해 이해한다.
- 리스트의 가장 중요한 특징인 인덱싱과 슬라이싱에 대해 학습한다.
- 리스트의 연산과 리스트를 추가하고 삭제하는 방법에 대해 알아본다.
- 패킹과 언패킹에 대해 알아보고, 이차원 리스트에 대해 이해한다.
- 리스트의 메모리 관리 방식에 대해 학습한다

01

파이썬 프로그래밍 환경

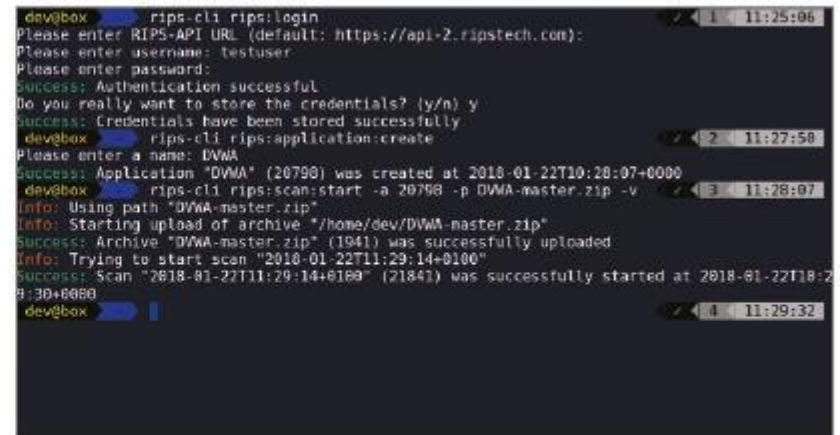
1. 사용자 인터페이스

■ 사용자 인터페이스(user interface)

- 컴퓨터에 명령을 입력할 때 사용하는 환경
- 대표적인 사용자 인터페이스: GUI(Graphical User Interface)



(a) GUI



(b) CLI

그림 3-1 여러 가지 사용자 인터페이스

2. CLI 환경

■ CLI(Command Line Interface) 환경

- 마우스의 클릭이 아닌 키보드만으로 명령을 입력하는 환경
- 매우 오래된 컴퓨터 사용자 인터페이스 체계
- 윈도우, 맥, 리눅스 등 모든 운영체제에서 기본으로 지원

TIP cmd 창: <윈도> + <R> → 'cmd' 입력 → <Enter> 클릭하면 나타나는 창

2. CLI 환경

여기서 잠깐! CLI 환경의 이름 유래

CLI 환경의 이름은 여러 가지다. 일반적으로 콘솔(console)이라고 부르지만 윈도우에서는 명령어 창(command window), 리눅스와 맥에서는 터미널(terminal)이라고 부른다. 그렇다면 '터미널'이라는 이름은 어디서 유래한 것일까?

이 이름의 유래는 1980년대 본격적인 개인용 컴퓨터(Personal Computer, PC) 시대가 되기 전까지의 컴퓨터 사용 환경에서 시작되었다. 초기 컴퓨터들은 지금처럼 개인용 컴퓨터가 아니라 거대한 고성능 메인프레임(mainframe)이 주로 사용되었다. 당연히 회사에 근무하지 않는 사람들은 컴퓨터를 만질 기회도 매우 제한적이었다. 물론 당시의 메인프레임 컴퓨터들은 지금의 스마트폰보다 성능이 떨어지지만, 여러 명이 이 컴퓨터를 쓰기 위해 접속을 해야 했다. 이때, 이 컴퓨터에 접속하기 위해 사용하는 머신의 이름을 '터미널'이라고 불렀다. 마치 버스 터미널과 같이 다른 곳으로 이동하기 위한 통로였던 것이다. 초기 터미널 머신은 당연히 텍스트로만 입력하는 CLI 환경이었다. 그러다 보니 이때 사용된 이름이 그대로 CLI 환경의 이름으로 굳어진 것이다. 당시 터미널은 키보드와 화면만 있고, 별도의 CPU나 저장장치는 모두 서버인 메인프레임의 것을 사용하였다. 프로그래머들은 실력이 늘수록, 또는 코드에 익숙해질수록 GUI보다 CLI 환경에 익숙해질 것이다.

02 화면 입출력

1. 표준 입력 함수: input() 함수

- 콘솔 창에 입력한 데이터를 간단하게 처리하는 프로그램을 작성하기
 - 파이썬에서는 콘솔 창에서 입력을 받기 위해 표준 입력 함수인 input() 함수 사용
 - Input() 함수를 이용해 사용자가 콘솔 창에서 문자열을 입력 받는 프로그램을 만들 수 있음

1. 표준 입력 함수: input() 함수

- 비주얼 스튜디오 코드에서 [코드 3-1] 작성하여 cmd 창에서 실행

[코드 3-1]

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

[실행결과]

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

1. 표준 입력 함수: input() 함수

TIP [코드 3-1]을 비주얼 스튜디오 코드에서 작성하고 'input.py'로 저장한 후, cmd 창에서 'python input.py'를 입력하면 코드를 실행할 수 있음.

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

그림 3-2 input.py를 cmd 창에서 실행

2. 표준 출력 함수: print() 함수

■ print() 함수 안에 있는 콤마(,)

- 여러 값을 연결해 화면에 출력 가능

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

```
Hello World! Hello Again!!!
```

콤마 사용

실행 시 두 문장이 연결되어 출력

- 콤마를 사용하면 'Hello World!'와 'Hello Again!!!' 사이에 한 칸 띄고 화면에 출력됨
- 비슷한 방법으로 문자형 간에 + 기호를 사용해 연결할 수 있음

2. 표준 출력 함수: print() 함수

■ input() 함수에 바로 지시문 넣기

[코드 3-2]

```
1 temperature = float(input("온도를 입력하세요: "))    # 입력 시 바로 형 변환
2 print(temperature)
```

[실행결과]

온도를 입력하세요: 103	← 입력 대기 및 사용자 입력
103.0	← 출력

3. 파일 입출력 정리

- 일반적으로 GUI 프로그램보다는 CLI 기반의 프로그램을 더 많이 개발하게 될 것임
- 파이썬의 태생적 근원이 스크립트 언어로서의 강력함에 있다 보니 콘솔 환경에 서 많이 사용하는 프로그램을 주로 작성하게 됨
- 특히 데이터 과학 분야에서는 매우 다양하게 사용될 수 있음

03

Lab: 화씨온도 변환기

- **화씨온도 변환기(Fahrenheit temperature converter) 프로그램:**

섭씨온도를 화씨온도로 변환시켜 주는 간단한 프로그램

- **섭씨온도(Celsius temperature):** 물의 어는점을 0°C, 끓는점을 100°C로 하여 이를 기준으로 삼고, 그 사이 간격을 100으로 나눈 온도
- **화씨온도(Fahrenheit temperature):** 물의 어는점을 32°F, 끓는점을 212°F로 하여 이를 기준으로 그 사이 간격을 180으로 나눈 온도

👉 이번 Lab에서는 섭씨온도를 입력하면 화씨온도로 바꿔주는 프로그램 작성

- 섭씨온도와 화씨온도의 변환 공식

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

[실행결과]

본 프로그램은 섭씨온도를 화씨온도로 변환하는 프로그램입니다.
변환하고 싶은 섭씨온도를 입력하세요.

32.2

섭씨온도: 32.2

화씨온도: 89.96

[코드 3-3]

```
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)
```

- 1행: `print()` 함수를 이용하여 `print("본 프로그램은 섭씨 온도를 화씨온도로 변환하는 프로그램입니다.")`를 입력.
- 2행: `print()` 함수를 이용하여 `print("변환하고 싶은 섭씨온도를 입력하세요.")`를 입력
- 4행: `input()` 함수를 사용, 사용자가 입력한 결과는 변수에 저장해야 하므로 `celsius`라는 변수 사용.
- 5행: 입력되는 값이 문자열이므로 실수형으로 변환하기 위해 `float()` 함수를 사용하여 자료형을 변환한 다음 입력된 섭씨온도 값을 화씨온도 값으로 변환.
- 7~8행: 결과를 출력하기 위해 `print()` 함수를 사용하여 코드를 작성하고 마무리.

04

리스트의 이해

1. 리스트가 필요한 이유

- 학생 100명의 성적을 채점해야 한다면 몇 개의 변수를 만들어야 할까?
 - 지금까지 배운 대로라면 100개의 변수를 만들어야 함.
 - 이렇게 되면 코드가 너무 길어지고 하나하나 입력해야 하고, 변수에 값을 하나씩 저장하는 것도 어려움
- ☞ 그래서 좀 더 쉬운 방법으로 한 개의 변수에 모든 값을 저장할 수 있음!
이러한 방식을 일반적으로는 배열, 파이썬에서는 리스트라고 함.

2. 리스트의 개념

■ 리스트 (list)

- **리스트:** 하나의 변수에 여러 값을 저장하는 자료형
- **시퀀스 자료형:** 리스트처럼 여러 데이터를 하나의 변수에 저장하는 기법을 파이썬에서 부르는 용어.
- 파이썬에서는 리스트라고 하지만 C나 자바 등에서는 '배열'이라고 더 많이 표현
- 파이썬에서도 배열의 개념이 있고 구분되어 사용하지만, 일반적으로 시퀀스 자료형을 처리하기 위해서는 리스트를 더 많이 사용

2. 리스트의 개념

- 'colors'라는 변수를 하나 만들고 리스트 자료형을 할당한 경우
 - 리스트의 할당은 대괄호를 사용해 리스트 안에 3개의 값(element)을 ['red', 'blue', 'green'] 형태로 저장
 - colors라는 변수는 3개의 값을 가지며, 각각의 값은 문자형의 red, blue, green이 됨.

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

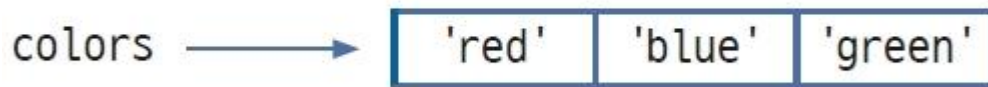


그림 3-3 리스트의 예

3. 인덱싱과 슬라이싱

3.1 인덱싱(indexing)

- 리스트에 저장되어 있는 값에 접근하기 위해 이 값의 상대적인 주소(offset)를 사용하는 것

[코드 3-4]

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

[실행결과]

```
red  
green  
3
```

3. 인덱싱과 슬라이싱

- colors라는 변수에는 'red', 'blue', 'green'이라는 값 3개가 저장되어 있음
- colors 변수가 리스트이므로 이 값들은 0번째부터 0, 1, 2라는 주소값으로 호출할 수 있음
- 2행의 결과로 colors 변수의 첫 번째 값인 'red'가 출력되고, 3행의 결과로 세 번째 값인 'green'이 출력됨
- 4행의 len() 함수는 리스트의 길이, 즉 리스트 안에 있는 값의 개수를 반환함

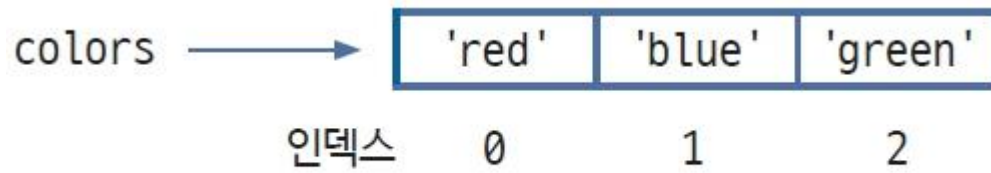



그림 3-4 리스트의 인덱싱

3. 인덱싱과 슬라이싱

여기서  잠깐! 리스트의 주소값은 왜 0부터 시작하는가?

대부분의 프로그래밍 언어에서 배열(array)과 같은 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. 현재는 큰 문제가 안 되지만, 초창기 컴퓨터들은 메모리 절약이 매우 큰 이슈였기 때문에 이 점은 아주 중요했다. 다만 비주얼 베이직이나 매트랩 같은 언어에서는 1부터 인덱싱을 한다는 것도 알아두면 좋다.

3. 인덱싱과 슬라이싱

3.2 슬라이싱(slicing)

- 리스트에서 파생된 강력한 기능 중 하나로, 리스트의 인덱스 기능을 사용하여 전체 리스트에서 일부를 잘라내어 사용하는 것.

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

- cities 변수에 8개의 도시 이름이 값으로 저장되어 있음.
- 0부터 7까지의 인덱스를 가진 총 8개의 값이 있는 것.

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

그림 3-5 cities 변수의 리스트

3. 인덱싱과 슬라이싱

- 슬라이싱의 기본 문법

변수명[시작 인덱스:마지막 인덱스]

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

- 파이썬의 리스트에서는 '마지막 인덱스 - 1'까지만 출력되고 마지막 인덱스값은 출력되지 않음.
- `cities[0:6]` - 0번째 인덱스값인 '서울'부터 6번째 인덱스값인 '울산'까지 나타낼 것으로 예상하기 쉽지만 마지막 '울산'은 출력되지 않음.

3. 인덱싱과 슬라이싱

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

- 한 번 이상 리스트 변수를 사용하면 마지막 인덱스가 다음 리스트의 시작 인덱스가 되어 코드를 작성할 때 조금 더 쉽게 이해할 수 있다는 장점이 있음.

3. 인덱싱과 슬라이싱

3.3 리버스 인덱스(reverse index)

- 기존 인덱스와 달리 마지막 값부터 -1을 할당하여 첫 번째 값까지 역순으로 올라오는 방식

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

그림 3-6 cities 변수의 리버스 인덱스

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

3. 인덱싱과 슬라이싱

3.4 인덱스 범위를 넘어가는 슬라이싱(slicing with over index)

- 슬라이싱을 할 때 인덱스의 첫 번째 값이나 마지막 값이 비어 있어도 잘 작동함.

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

- 인덱스를 넣지 않고 `cities[:]`과 같이 콜론(:)을 넣으면 `cities` 변수의 모든 값을 반환함.
- `Cities[-50:50]`처럼 범위를 넘어가는 인덱스를 입력하는 경우도 동일하게 해당 리스트의 시작부터 끝까지의 인덱스값으로 데이터를 불러옴.

3. 인덱싱과 슬라이싱

3.5 증가값(step)

- 슬라이싱에서는 시작 인덱스와 마지막 인덱스 외에도 마지막 자리에 증가값을 넣을 수 있음.

변수명[시작 인덱스:마지막 인덱스:증가값]

```
>>> 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']
```

- `cities[::2]`를 입력하면 처음부터 시작하여 2칸 간격으로 출력
- `cities[::-1]`은 역방향으로 1칸 간격으로 출력

4. 리스트의 연산

4.1 덧셈 연산

```
>>> 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
>>>
```

- color1과 color2라는 리스트 변수를 만들고 덧셈 연산으로 두 변수를 합치면, 각각의 리스트가 하나의 리스트로 합쳐져서 출력됨.
- 덧셈 연산을 하더라도 따로 어딘가에 변수 형태로 저장해주지 않으면 기존 변수들에는 아무 변화가 없음.

4. 리스트의 연산

- 맨 마지막 코드에 있는 total_color를 출력하면 다음과 같이 합쳐진 값이 저장되어 있음.

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

4. 리스트의 연산

4.2 곱셈 연산

- 리스트에 n 을 곱했을 때 해당 리스트를 n 배만큼 늘려줌.

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

- 위 코드를 보면 color1 리스트가 2번 반복되어 출력되는 것을 알 수 있음.

4.3 in 연산

- 포함 여부를 확인하는 연산으로 하나의 값이 해당 리스트에 들어있는지 확인할 수 있음.

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

5. 리스트 추가 및 삭제

5.1 append() 함수

- append() 함수를 사용하면 리스트 맨 마지막 인덱스에 새로운 값을 추가할 수 있음

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

5.2 extend() 함수

- 값을 추가하는 것이 아닌 기존 리스트에 그대로 새로운 리스트를 합치는 기능으로 리스트의 덧셈 연산과 같음.

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

5. 리스트 추가 및 삭제

5.3 insert() 함수

- append() 함수와 달리 리스트의 특정 위치에 값을 추가함

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

5.4 remove() 함수

- 리스트에 있는 특정 값을 지우는 기능
- 삭제할 값을 remove() 함수 안에 넣으면 리스트에 있는 해당 값이 삭제됨.

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

5. 리스트 추가 및 삭제

5.5 인덱스의 재할당과 삭제

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

- 특정 인덱스 값을 변경하기 위해서는 인덱스에 새로운 값을 할당하면 됨.
 - 특정 인덱스 값을 삭제하기 위해서는 del 함수를 사용
- 👉 `del color[0]` 0번째 값을 삭제하라고 입력하면, 0번째 인덱스에 있던 'orange'가 삭제됨.

5. 리스트 추가 및 삭제

표 3-1 리스트 추가 및 삭제 함수

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]

6. 패킹과 언패킹

```
>>> 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
```

- **패킹:** 한 변수에 여러 개의 데이터를 할당하는 그 자체로 리스트 자체를 뜻하기도 함.
- **언패킹:** 한 변수에 여러 개의 데이터가 들어있을 때 그것을 각각의 변수로 반환하는 방법.

6. 패킹과 언패킹

- 리스트에 값이 3개인데, 5개로 언패킹을 시도한다면 어떤 결과가 나올까?

```
>>> 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)
```

☞ 언패킹 시 할당 받는 변수의 개수가 적거나 많으면 모두 에러가 발생.

6. 이차원 리스트

- **이차원 리스트:** 표에 값을 채웠을 때 생기는 값들의 집합
- 리스트를 효율적으로 활용하기 위해 여러 개의 리스트를 하나의 변수에 할당하는 이차원 리스트를 사용할 수 있음.

표 3-2 이차원 리스트의 예

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

6. 이차원 리스트

- [표 3-2]와 같은 이차원 리스트를 하나의 변수로 표현하기

```
>>> 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]]
```

- 국어 점수, 수학 점수, 영어 점수를 각각 kor_score, math_score, eng_score에 할당하고, 그 각각의 변수를 모두 midterm_score에 할당함.

- 이차원 리스트 값에 접근하기 위해서는 대괄호 2개를 사용하여 인덱싱함.

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

- 이차원 리스트를 행렬로 본다면 [0]은 행, [2]는 열을 뜻함.

05

리스트의 메모리 관리 방식

1. 리스트의 메모리 저장

```
>>> 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]]
```

- 위 코드의 가장 핵심은 `math_score[0] = 1000` 임.
- `math_score`의 값을 변경하였는데 `midterm_score` 두 번째 행의 첫 번째 값이 변경된 이유는?
 - ☞ 파이썬 리스트가 값을 저장하는 방식 때문에 발생하는 현상임.
 - ☞ 파이썬은 리스트를 저장할 때 값 자체가 아니라 값이 위치한 메모리 주소를 저장하기 때문.

1. 리스트의 메모리 저장

- 리스트 안에는 값 자체를 저장하는 구조가 아니라 그 값이 위치한 메모리의 주소, 즉 '0x3172'와 같은 주소 값을 저장함.

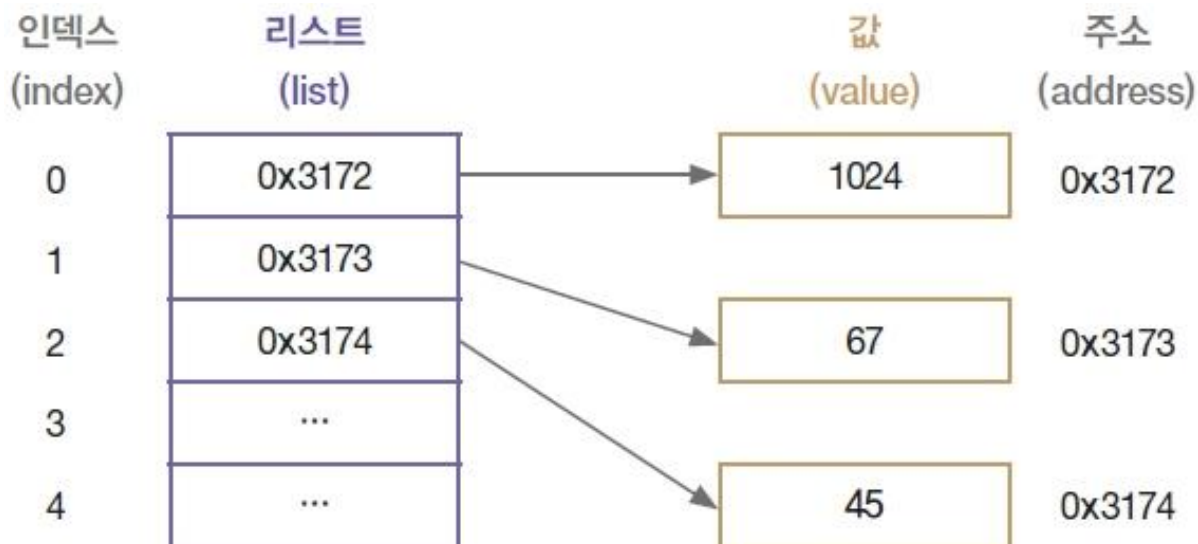


그림 3-7 리스트의 메모리 저장

1. 리스트의 메모리 저장

- 값과 메모리 주소 값의 차이

```
>>> a = 300
>>> b = 300
>>> a is b
False
>>> a == b
True
```

- 위 코드에서 a와 b라는 변수에 모두 300이라는 값을 할당함. 즉, 변수 a와 b에 저장되어 있는 메모리 주소를 따라가면 값 300이 저장되어 있는 것.
- 그런데 왜 ==로 비교하면 True 인데, is로 비교하면 False가 나올까?
☞ ==은 값을 비교하는 연산이고, is는 메모리의 주소를 비교하는 연산이기 때문.

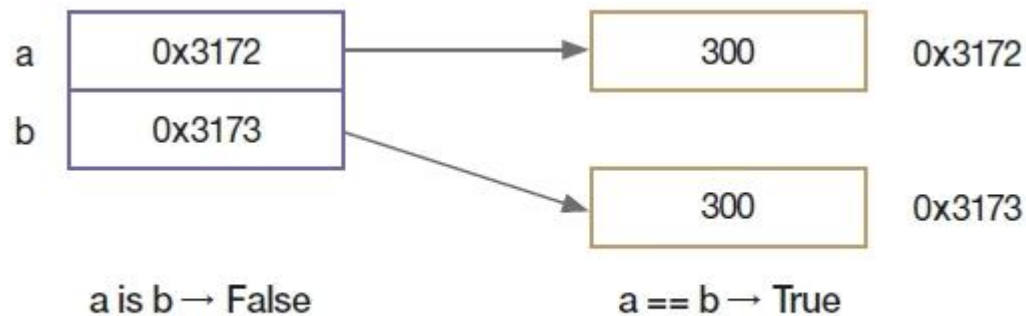


그림 3-8 변수 a와 b의 값과 메모리 주소의 차이

1. 리스트의 메모리 저장

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

- 이전 코드와 다르게 is와 == 연산자는 모두 True를 반환한 이유는? 파이썬의 정수형 저장 방식의 특성 때문.
- 파이썬은 인터프리터가 구동될 때, -5부터 256까지의 정수값을 특정 메모리 주소에 저장하고 해당 숫자를 할당하려고 할 때 해당 변수는 그 숫자가 가진 메모리 주소로 연결함. 👉 따라서 주소와 값이 모두 같은 것으로 나오는 것.
- 리스트는 기본적으로 값을 연속으로 저장하는 것이 아니라, 값이 있는 주소를 저장하는 방식임.

1. 리스트의 메모리 저장

- 제일 처음 나온 예제 코드에서 midterm_score 변수에는 math_score의 메모리 주소를 가지고 있기 때문에 math_score의 값이 변하면 midterm_score 주소값이 가르키고 있는 변경된 값을 보여줌.

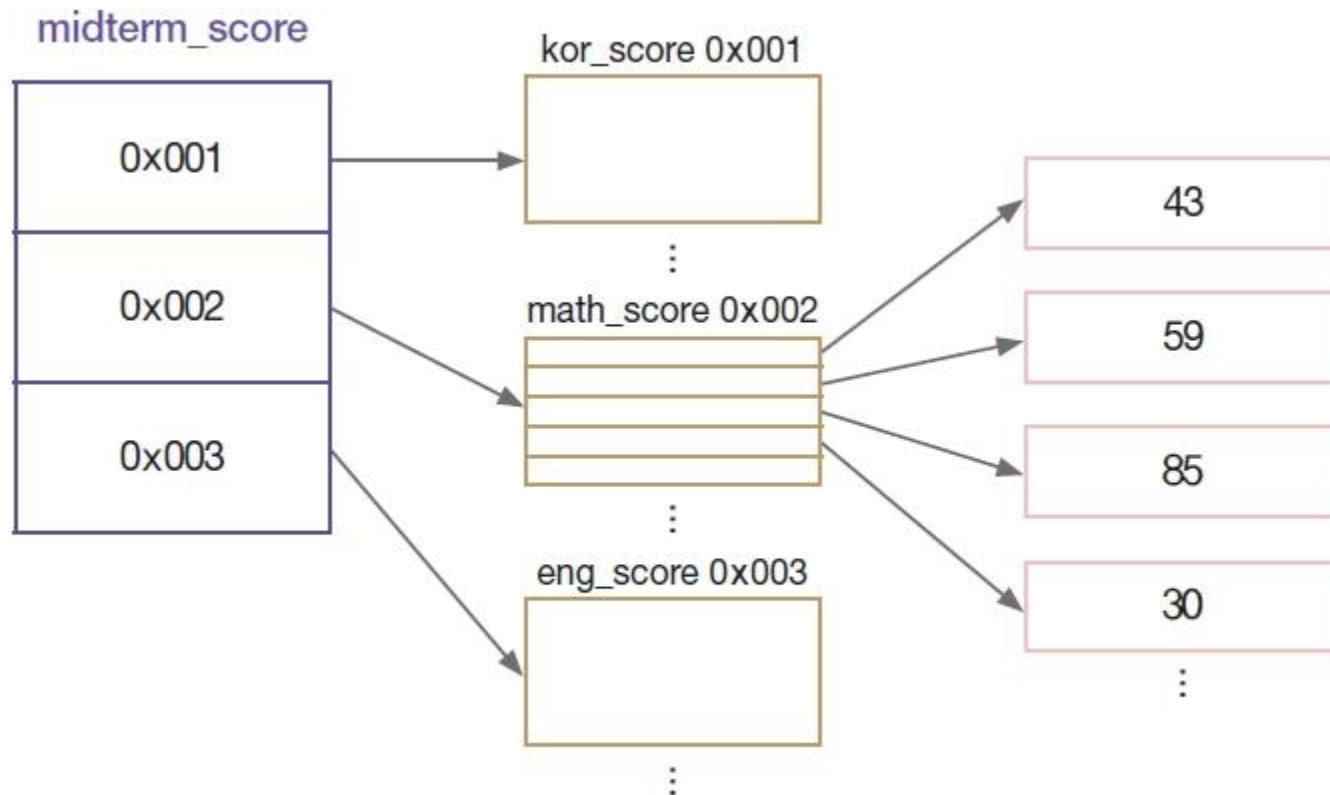


그림 3-9 리스트의 메모리 저장 연결 관계

2. 메모리 저장 구조로 인한 리스트의 특징

2.1 하나의 리스트에 다양한 자료형 포함 가능

- 첫 번째 리스트의 특징: 다양한 자료형이 하나의 리스트에 모두 들어갈 수 있음.

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

2. 메모리 저장 구조로 인한 리스트의 특징

- 기존 변수들과 함께 리스트 안에 다른 리스트를 넣을 수도 있음.

```
>>> 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]
```

- 다음 코드에서 color라는 리스트 변수를 새로 만들고 이를 a에 0번째 인덱스에 값으로 넣어도 문제없이 할당됨
 - 즉 "color"라는 문자열 대신에 color 리스트를 넣으라는 뜻임.
- ☞ 이것을 중첩 리스트라고 함

2. 메모리 저장 구조로 인한 리스트의 특징

2.2 리스트의 저장 방식

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

- b와 a 변수를 각각 다른 값으로 선언한 후 b에 a를 할당하고 b를 출력하면 a 변수와 같은 값이 화면에 출력됨.

- a만 정렬하고 b를 출력하면 어떤 결과가 나올까?

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

TIP sort() 함수: 리스트에 있는 값들의 순서를 오름차순으로 변환하는 함수

2. 메모리 저장 구조로 인한 리스트의 특징

- a를 정렬(sorting)했는데 b도 정렬됨

☞ b = a를 입력하는 순간 b에도 a 리스트의 메모리 주소가 저장되기 때문

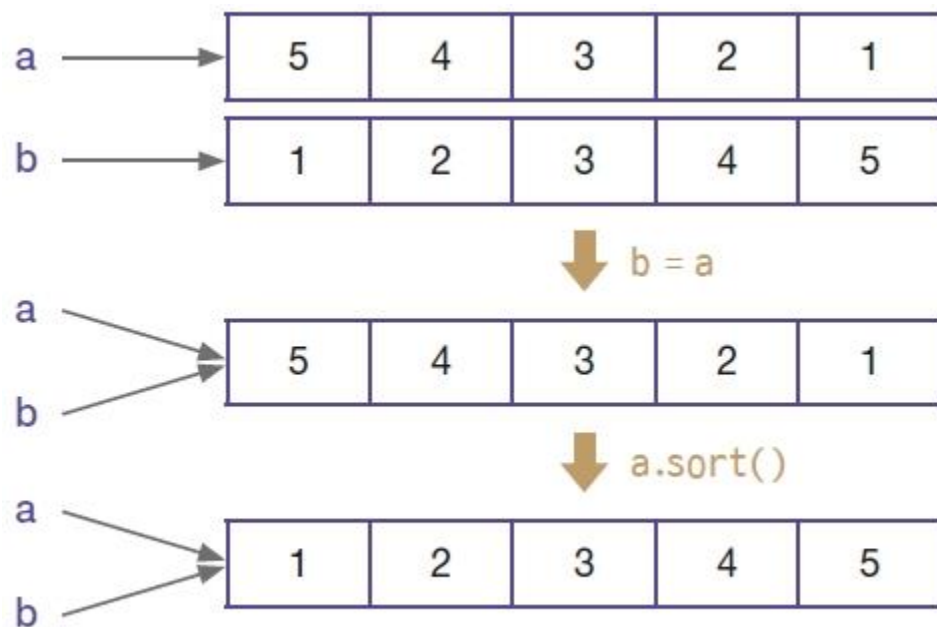


그림 3-10 a를 정렬했는데 b도 정렬되는 이유

2. 메모리 저장 구조로 인한 리스트의 특징

- b에 새로운 값을 할당하면 어떤 변화가 나타날까?

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

- b에 새로운 값을 할당하면 a와 b는 다른 메모리 주소를 할당 받음. 즉, b는 이제 새로운 메모리 주소가 가리키는 곳에 값을 할당할 수 있는 것.
- `b = a 코드` ➡ 어떤 리스트 값을 하나의 변수에 할당하는 순간 두 변수는 같은 메모리 주소를 할당받게 됨.

TIP '='의 의미는 같다가 아닌 메모리 주소에 해당 값을 할당(연결)한다는 의미

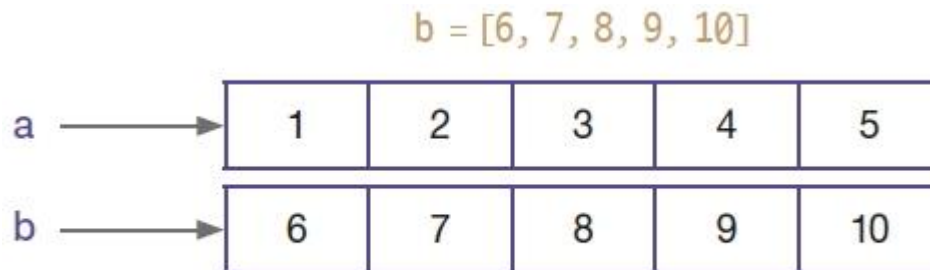


그림 3-11 b에 새로운 값을 할당하는 경우

Thank you!