Section01 자료구조

▪ 자료구조의 개념

- **자료구조(data structure) :** 특징이 있는 정보를 메모리에 효율적으로 저장 및 반환하는 방법으로, 데이터를 관리하는 방식이다. 특히 대용량일수록 메모리에 빨리 저장하고 빠르게 검색하여, 메모리를 효율적으로 사용하고 실행 시간을 줄일 수 있게 해 준다.

(a) 전화번호부 (b) 휴대전화의 연락처

[실생활 속 자료구조]

Section01 자료구조

▪ 파이썬에서의 자료구조

자료구조명	특징
스택(stack)	나중에 들어온 값을 먼저 나갈 수 있도록 해 주는 자료구조(last in first out)
큐(queue)	먼저 들어온 값을 먼저 나갈 수 있도록 해 주는 자료구조(first in first out)
튜플(tuple)	리스트와 같지만, 데이터의 변경을 허용하지 않는 자료구조
세트(set)	데이터의 중복을 허용하지 않고, 수학의 집합 연산을 지원하는 자료구조
딕셔너리 (dictionary)	전화번호부와 같이 키(key)와 값(value) 형태의 데이터를 저장하는 자료구조. 여기서 키값은 다른 데이터와 중복을 허용하지 않음
collections 모듈	위에 열거된 여러 자료구조를 효율적으로 사용할 수 있도록 지원하는 파이썬 내장(built-in) 모듈

[파이썬에서 제공하는 자료구조]

Section02 리스트의 기본

▪ 리스트의 개념

▪ 딕셔너리를 활용해 음식 궁합을 출력하는 프로그램

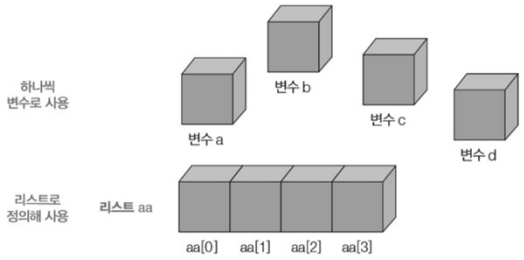


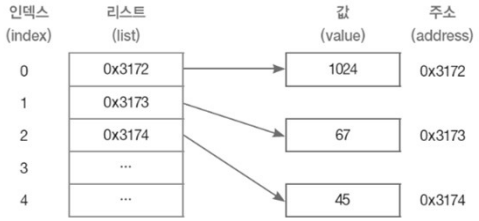
그림 7-1 리스트의 개념

Tip • C/C++나 자바 같은 프로그래밍 언어에는 리스트가 없음, 리스트와 비슷한 개념인 배열(Array)을 사용. 리스트는 정수, 문자열, 실수 등 서로 다른 데이터형도 하나로 묶을 수 있지만, 배열은 동일한 데이터형만 묶을 수 있다. 정수 배열은 정수로만 묶어서 사용

Section02 리스트의 기본

▪ 리스트의 메모리 저장

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

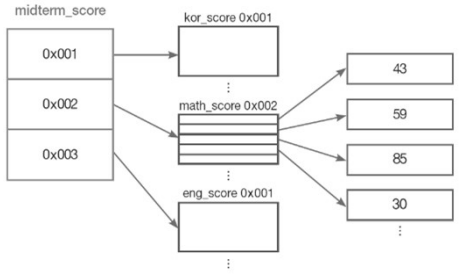


[리스트의 메모리 저장]

Section02 리스트의 기본

▪ 리스트의 메모리 저장

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



Section02 리스트의 기본

▪ 리스트의 필요성

▪ a, b, c, d라는 정수형 변수 선언 후 이 변수에 값을 입력받고 합계 출력하는 프로그램

Code07-01.py

```
1  a, b, c, d = 0, 0, 0, 0
2  hap = 0
3
4  a = int(input("1번째 숫자 : "))
5  b = int(input("2번째 숫자 : "))
6  c = int(input("3번째 숫자 : "))
7  d = int(input("4번째 숫자 : "))
8
9  hap = a + b + c + d
10
11 print("합계 ==> %d" % hap)
```

출력 결과

```
1번째 숫자 : 10
2번째 숫자 : 20
3번째 숫자 : 30
4번째 숫자 : 40
합계 ==> 100
```


Section02 리스트의 기본

▪ 리스트를 생성하는 방법

```
리스트명 = [값1, 값2, 값3, …]
```

```
aa = [10, 20, 30, 40]
```

❶ 각 변수 사용	❷ 리스트 사용
a, b, c, d = 10, 20, 30, 40	aa = [10, 20, 30, 40]
a 사용	aa[0] 사용
b 사용	aa[1] 사용
c 사용	aa[2] 사용
d 사용	aa[3] 사용

Section02 리스트의 기본

▪ Code07-01.py를 리스트를 사용하는 프로그램으로 수정

Code07-02.py

```
1  aa = [0, 0, 0, 0]
2  hap = 0
3
4  aa[0] = int(input("1번째 숫자 : "))
5  aa[1] = int(input("2번째 숫자 : "))
6  aa[2] = int(input("3번째 숫자 : "))
7  aa[3] = int(input("4번째 숫자 : "))
8
9  hap = aa[0] + aa[1] + aa[2] + aa[3]
10
11 print("합계 ==> %d" % hap)
```

1행 : aa=[0, 0, 0, 0]으로 항목이 4개 있는 리스트 생성
4행 : a 대신 aa[0]을 사용
5~7행 : 리스트 aa를 사용
9행 : 각 변수 대신 aa[0]+aa[1]+aa[2]+aa[3]으로 수정

출력 결과

```
1번째 숫자 : 10
2번째 숫자 : 20
3번째 숫자 : 30
4번째 숫자 : 40
합계 ==> 100
```

출력 결과는 리스트를 사용하기 전과 동일
숫자 100개를 더하려면 aa=[0, 1, 0, ..., 99]를 생성한 후
aa[0]+aa[1]+aa[2] +...+aa[99]로 작성

Section02 리스트의 기본

- **리스트의 일반적인 사용**
  - 빈 리스트의 생성과 항목 추가

```
aa = []
aa.append(0)
aa.append(0)
aa.append(0)
aa.append(0)
print(aa)
```

출력 결과

```
[0, 0, 0, 0]
```

```
aa = []
for i in range(0, 100) :
    aa.append(0)
len(aa)
```

출력 결과

```
100
```


Section02 리스트의 기본

- 리스트의 첨자가 순서대로 변할 수 있도록 반복문과 함께 활용

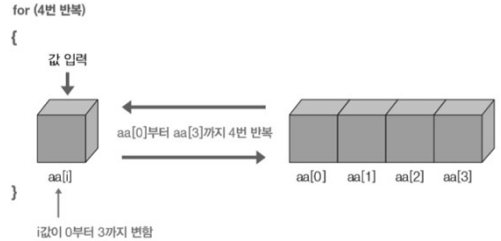


그림 7-2 for 문으로 리스트값 입력

Section02 리스트의 기본

Code07-03.py

```
1  aa = []
2  for i in range(0, 4) :
3      aa.append(0)
4  hap = 0
5
6  for i in range(0, 4) :
7      aa[i] = int(input(str(i+1) + '번째 숫자를 입력 : '))
8
9  hap = aa[0] + aa[1] + aa[2] + aa[3]
10
11 print("합계 ==> %d" % hap)
```

1행 : 빈 리스트 생성
2~3행 : 4번을 반복해 항목이 4개인 리스트로 만듦
6행 : i가 0에서 3까지 4번 반복

7행 : input() 함수는 첨자 i가 0부터 시작하므로 i+1로 출력. str() 함수가 정수를 문자로 변환한 후 '번째 숫자 : '와 합쳐지므로 결국 '1번째 숫 자 : ', '2번째 숫자 : ' 등으로 출력. 7행의 첨자 i가 0에서 3까지 4번 변경되므로 aa[0], aa[1], aa[2], aa[3] 등 변수 4개에 값을 차례대로 입력해 [그림 7-2]와 같이 작동
9행 : 변수 4개를 더함

출력 결과

```
1번째 숫자 : 10
2번째 숫자 : 20
3번째 숫자 : 30
4번째 숫자 : 40
합계 ==> 100
```


Section02 리스트의 기본

- 9행의 변경

```
for i in range(0, 4) :
    hap = hap + aa[i]
```

SELF STUDY 7-1

값을 4개가 아닌 10개를 입력받아 합계를 출력하도록 Code07-03.py를 수정해 보자. 또 합계를 구하는 마지막 for 문 대신 while 문을 사용해 보자.

Section02 리스트의 기본

- **리스트의 생성과 초기화**
  - 리스트 생성 코드

```
❶ aa = []
❷ bb = [10, 20, 30]
❸ cc = ['파이썬', '공부는', '꿀잼']
❹ dd = [10, 20, '파이썬']
```

❶ 빈 리스트 생성
❷ 정수로만 구성된 리스트 생성
❸ 문자열로만 구성된 리스트 생성
❹ 다양한 데이터형을 섞어 리스트 생성

  - 리스트 100개인 aa 0, 2, 4, 8, ...(2의 배수로 초기화 후 리스트 bb에 역순으로 넣는 과정

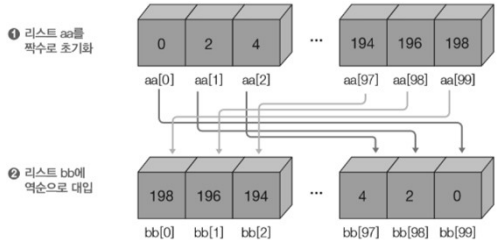


그림 7-3 리스트의 초기화 및 역순 대입

Section02 리스트의 기본

- [그림 7-3] 코드로 구현

Code07-04.py

```
1  aa = []
2  bb = []
3  value = 0
4
5  for i in range(0, 100) :
6      aa.append(value)
7      value += 2
8
9  for i in range(0, 100) :
10     bb.append(aa[99 - i])
11
12 print("bb[0]에는 %d이, bb[99]에는 %d이 입력됩니다." % (bb[0], bb[99]))
```

1~2행 : 빈 리스트 aa, bb 생성
3행 : value는 0, 2, 4, ...로 증가시킬 값
5행 : 100번을 반복
6~7행 : 리스트 aa에 value를 추가한 후 2씩 증가
9행 : 0~99로 100번 반복
10행 : i가 0일 때 99-i는 99가 됨. i가 1일 때는 98, i가 2일 때 는 97처럼 계속 변해 마지막으로 i가 99일 때는 0이 됨.
리스트 bb에는 aa[99], aa[98], aa[97], ..., aa[0]의 값이 추가되므로 결국 리스트 aa값이 리스트 bb에 역순으로 입력
12행 : 확인을 하려고 bb[0]과 bb[99]를 출력

출력 결과

```
bb[0]에는 198이, bb[99]에는 0이 입력됩니다.
```


Section02 리스트의 기본

- **리스트 값에 접근하는 다양한 방법**
  - 음수값으로 접근

```
aa = [10, 20, 30, 40]
print("aa[-1]은 %d, aa[-2]는 %d" % (aa[-1], aa[-2]))
```

출력 결과

```
aa[-1]은 40, aa[-2]는 30
```

aa[-4]까지 접근되므로 aa[-5]는 오류 발생

  - 리스트에 접근할 때 콜론(:)을 사용해 범위를 지정

```
aa = [10, 20, 30, 40]
aa[0:3]
aa[2:4]
```

출력 결과

```
[10, 20, 30]
[30, 40]
```


Section02 리스트의 기본

- 콜론의 앞이나 뒤 숫자 생략

```
aa = [10, 20, 30, 40]
aa[2:]
aa[:2]
```

출력 결과

```
[30, 40]
[10, 20]
```

- 리스트끼리 덧셈, 곱셈 연산

```
aa = [10, 20, 30]
bb = [40, 50, 60]
aa + bb
aa * 3
```

출력 결과

```
[10, 20, 30, 40, 50, 60]
[10, 20, 30, 10, 20, 30, 10, 20, 30]
```

Section02 리스트의 기본

- 리스트의 항목 건너뛰며 추출

```
aa = [10, 20, 30, 40, 50, 60, 70]
aa[::2]
aa[::-2]
aa[::-1]
```

출력 결과

```
[10, 30, 50, 70]
[70, 50, 30, 10]
[70, 60, 50, 40, 30, 20, 10]
```

17/59

Section02 리스트의 기본

- **리스트 값의 변경**
  - 두 번째에 위치한 값을 1개 변경하는 방법

```
aa = [10, 20, 30]
aa[1] = 200
aa
```

출력 결과

```
[10, 200, 30]
```

  - 두 번째 값 인 20을 200과 201이라는 값 2개로 변경하는 방법

```
aa = [10, 20, 30]
aa[1:2] = [200, 201]
aa
```

출력 결과

```
[10, 200, 201, 30]
```

18/59

Section02 리스트의 기본

- aa[1:2] 대신 그냥 aa[1] 사용

```
aa = [10, 20, 30]
aa[1] = [200, 201]
aa
```

출력 결과

```
[10, [200, 201], 30]
```

- 두 번째인 aa[1]의 항목 삭제

```
aa = [10, 20, 30]
del(aa[1])
aa
```

출력 결과

```
[10, 30]
```

19/59

Section02 리스트의 기본

- 두 번째인 aa[1]에 서 네 번째인 aa[3]까지 삭제

```
aa = [10, 20, 30, 40, 50]
aa[1:4] = []
aa
```

출력 결과

```
[10, 50]
```

- 리스트 자체를 삭제하는 방법

```
❶ aa = [10, 20, 30]; aa = []; aa
❷ aa = [10, 20, 30]; aa = None; aa
❸ aa = [10, 20, 30]; del(aa); aa
```

출력 결과

```
[]
아무것도 안 나옴
오류 발생
```

20/59

Section02 리스트의 기본

- **리스트 조작 함수**

표 7-1 **리스트 조작 함수**

함수	설명	사용법
append()	리스트 맨 뒤에 항목을 추가한다.	리스트명.append(값)
pop()	리스트 맨 뒤의 항목을 빼낸다(리스트에서 해당 항목이 삭제된다).	리스트명.pop()
sort()	리스트의 항목을 정렬한다.	리스트명.sort()
reverse()	리스트 항목의 순서를 역순으로 만든다.	리스트명.reverse()
index()	지정한 값을 찾아 해당 위치를 반환한다.	리스트명.index(찾을값)
insert()	지정된 위치에 값을 삽입한다.	리스트명.insert(위치, 값)
remove()	리스트에서 지정한 값을 삭제한다. 단 지정한 값이 여러 개면 첫 번째 값만 지운다.	리스트명.remove(지울값)
extend()	리스트 뒤에 리스트를 추가한다. 리스트의 더하기(+) 연산과 기능이 동일하다.	리스트명.extend(추가할리스트)
count()	리스트에서 해당 값의 개수를 센다.	리스트명.count(찾을값)
clear()	리스트의 내용을 모두 지운다.	리스트명.clear()
del()	리스트에서 해당 위치의 항목을 삭제한다.	del(리스트명[위치])
len()	리스트에 포함된 전체 항목의 개수를 센다.	len(리스트명)
copy()	리스트의 내용을 새로운 리스트에 복사한다.	새리스트=리스트명.copy()
sorted()	리스트의 항목을 정렬해서 새로운 리스트에 대입한다.	새리스트=sorted(리스트)

Section02 리스트의 기본

- 예

Code07-05.py

```
1  myList = [30, 10, 20]
2  print("현재 리스트 : %s" % myList)
3
4  myList.append(40)
5  print("append(40) 후의 리스트 : %s" % myList)
6
7  print("pop()으로 추출한 값 : %s" % myList.pop())
8  print("pop() 후의 리스트 : %s" % myList)
9
10 myList.sort()
11 print("sort() 후의 리스트 : %s" % myList)
12
13 myList.reverse()
14 print("reverse() 후의 리스트 : %s" % myList)
15
16 print("20값의 위치 : %d" % myList.index(20))
17
18 myList.insert(2, 222)
19 print("insert(2, 222) 후의 리스트 : %s" % myList)
20
```

10행 : '리스트.sort()'는 리스트 자체 정렬 'sorted(리스트)'는 리스트는 그대로 두고 정렬된 결과만 반환

18행 : myList.insert(2, 222)에서 2는 myList[2]의 위치를 의미, 리스트는 0번부터 시작 하므로 세 번째 위치가 뒤로 밀리고 그 자리에 222가 삽입

Section02 리스트의 기본

```
21 myList.remove(222)
22 print("remove(222) 후의 리스트 : %s" % myList)
23
24 myList.extend([77, 88, 77])
25 print("extend([77, 88, 77]) 후의 리스트 : %s" % myList)
26
27 print("77값의 개수 : %d" % myList.count(77))
```

출력 결과

```
현재 리스트 : [30, 10, 20]
append(40) 후의 리스트 : [30, 10, 20, 40]
pop()으로 추출한 값 : 40
pop() 후의 리스트 : [30, 10, 20]
sort() 후의 리스트 : [10, 20, 30]
reverse() 후의 리스트 : [30, 20, 10]
20값의 위치 : 1
insert(2, 222) 후의 리스트 : [30, 20, 222, 10]
remove(222) 후의 리스트 : [30, 20, 10]
extend([77, 88, 77]) 후의 리스트 : [30, 20, 10, 77, 88, 77]
77값의 개수 : 2
```


Section02 리스트의 기본

- 기존 리스트는 변경하지 않고 정렬된 새로운 리스트 생성

```
myList = [30, 10, 20]
newList = sorted(myList)
print("sorted() 후의 myList : %s" % myList)
print("sorted() 후의 newList : %s" % newList)
```

출력 결과

```
sorted() 후의 myList : [30, 10, 20]
sorted() 후의 newList : [10, 20, 30]
```

Section03 2차원 리스트

- **2차원 리스트의 개념**
  - 1차원 리스트를 여러 개 연결한 것, 첨자를 2개 사용

```
aa = [10, 20, 30]
```

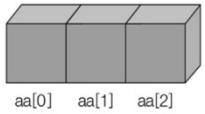


그림 7-4 **1차원 리스트의 개념**

```
aa = [[1, 2, 3, 4],
      [5, 6, 7, 8],
      [9, 10, 11, 12]]
```

그림 7-5 **2차원 리스트의 개념**

Section03 2차원 리스트

- 예 : 중첩 for 문을 사용, 3행 4열짜리 리스트 생성 후 항목 1~12를 입력하고 출력

Code07-06.py

```
1  list1 = []
2  list2 = []
3  value = 1
4  for i in range(0, 3) :
5      for k in range(0, 4) :
6          list1.append(value)
7          value += 1
8      list2.append(list1)
9      list1 = []
10
11 for i in range(0, 3) :
12     for k in range(0, 4) :
13         print("%3d" % list2[i][k], end = " ")
14     print("")
```

1~2행 : 1차원 리스트로 사용할 list1과 2차원 리스트로 사용할 list2 준비
3행 : value는 리스트에 입력할 1~12의 값으로 사용할 변수
4~9행 : 리스트의 행 단위 만들기 위해 3회 반복
5~7행 : 4회 반복해 항목 4개인 1차원 리스트 생성하는데, 처음에는 [1, 2, 3, 4]의 리스트를 만듬
8행 : 2차원 리스트에 추가
9행 : 1차원 리스트를 다시 비움
11~14행 : 2차원 리스트 출력
13행 : '리스트명[행][열]' 방식으로 각 항목 출력
14행 : 행 하나 출력 후 한 행 띄워서 출력 위해 print() 문 사용

출력 결과

```
 1  2  3  4
 5  6  7  8
 9 10 11 12
```


Section03 2차원 리스트

SELF STUDY 7-3

4행 5열의 2차원 리스트를 만들고, 0부터 3의 배수를 입력하고 출력하도록 Code07-06.py를 수정해 보자. 출력 결과는 다음과 같다.

출력 결과

```
 0  3  6  9 12
15 18 21 24 27
30 33 36 39 42
45 48 51 54 57
```


Section03 2차원 리스트

- 불규칙한 크기의 2차원 리스트

```
aa = [[1, 2, 3, 4],
      [5, 6],
      [7, 8, 9]]
```

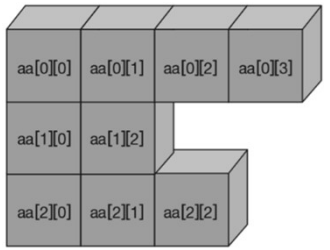


그림 7-6 **불규칙한 크기의 2차원 리스트**

Section04 튜플

- **튜플의 생성**
  - 리스트는 대괄호 []로 생성, 튜플은 소괄호 ()로 생성
  - 튜플은 값을 수정할 수 없으며, 읽기만 가능해 읽기 전용 자료를 저장할 때 사용

```
tt1 = (10, 20, 30); tt1
tt2 = 10, 20, 30; tt2
```

출력 결과

```
(10, 20, 30)
(10, 20, 30)
```

  - 튜플은 소괄호 ()를 생략 가능, 항목이 하나인 튜플은 tt5와 tt6처럼 뒤에 쉼표(,) 붙임

```
tt3 = (10); tt3
tt4 = 10; tt4
tt5 = (10,); tt5
tt6 = 10,; tt6
```

출력 결과

```
10
10
(10,)
(10,)
```

29/59

Section04 튜플

- 튜플의 오류

```
tt1.append(40)
tt1[0] = 40
del(tt1[0])
```

- 튜플의 삭제

```
del(tt1)
del(tt2)
```

30/59

Section04 튜플

- **튜플의 사용**
  - 튜플 항목에 접근

```
tt1 = (10, 20, 30, 40)
tt1[0]
tt1[0] + tt1[1] + tt1[2]
```

출력 결과

```
10
60
```

  - 튜플 범위에 접근

```
tt1[1:3]
tt1[1:]
tt1[:3]
```

출력 결과

```
(20, 30)
(20, 30, 40)
(10, 20, 30)
```

31/59

Section04 튜플

- 튜플의 덧셈 및 곱셈 연산

```
tt2 = ('A', 'B')
tt1 + tt2
tt2 * 3
```

출력 결과

```
(10, 20, 30, 40, 'A', 'B')
('A', 'B', 'A', 'B', 'A', 'B')
```

SELF STUDY 7-4

다음과 같이 2차원 튜플을 생성한 후 모든 값을 출력해 보자.

```
tt = ((1, 2, 3),
      (4, 5, 6),
      (7, 8, 9))
```

출력 결과

```
1 2 3
4 5 6
7 8 9
```

32/59

Section04 튜플

- 예 : 튜플 → 리스트 → 튜플 변환

```
myTuple = (10, 20, 30)
myList = list(myTuple)
myList.append(40)
myTuple = tuple(myList)
myTuple
```

출력 결과

```
(10, 20, 30, 40)
```

33/59

Section05 딕셔너리

■ 딕셔너리의 개념

- 쌍 2개가 하나로 묶인 자료구조
  - 예 : 'apple:사과'처럼 의미 있는 두 값을 연결해 구성

Tip • 다른 프로그래밍 언어에서는 해시(Hash), 연관 배열(Associative Array)이라 함

- 중괄호 {}로 묶어 구성, 키(Key)와 값(Value)의 쌍으로 구성

```
딕셔너리변수 = {키1:값1, 키2:값2, 키3:값3, …}
```

학번	이름	생년월일	주소
20150230	홍길동	1995-04-03	서울시 동대문구
20150233	김영철	1995-04-20	성남시 분당구
20150234	오영심	1996-01-03	성남시 중원구
20150236	최성철	1995-12-27	인천시 계양구

[대학생 인적사항]

34/59

Section05 딕셔너리

■ 딕셔너리의 생성

```
dic1 = {1 : 'a', 2 : 'b', 3 : 'c'}
dic1
```

출력 결과

```
{1 : 'a', 2 : 'b', 3 : 'c'}
```

- 키와 값을 반대로

```
dic2 = {'a': 1, 'b': 2, 'c': 3}
dic2
```

출력 결과

```
{'a': 1, 'b': 2, 'c': 3}
```

- 키와 값은 사용자가 지정하는 것이지 규정은 없음
- 주의할 점 : 딕셔너리에는 순서가 없어 생성한 순서대로 딕셔너리가 구성되어 있다는 보장 없음

35/59

Section05 딕셔너리

- 여러 정보의 딕셔너리 표현

표 7-2 **홍길동 학생의 정보**

키	값
학번	1000
이름	홍길동
학과	컴퓨터학과

```
student1 = {'학번' : 1000, '이름': '홍길동', '학과': '컴퓨터학과'}
student1
```

출력 결과

```
{'학번': 1000, '이름': '홍길동', '학과': '컴퓨터학과'}
```

- student1에 연락 처 추가

```
student1['연락처'] = '010-1111-2222'
student1
```

출력 결과

```
{'학번': 1000, '이름': '홍길동', '학과': '컴퓨터학과', '연락처': '010-1111-2222'}
```

36/59

Section05 딕셔너리

- 학과 수정

```
student1['학과'] = '파이썬학과'
student1
```

출력 결과

```
{'학번': 1000, '이름': '홍길동', '학과': '파이썬학과', '연락처': '010-1111-2222'}
```

- student1의 학과 삭제

```
del(student1['학과'])
student1
```

출력 결과

```
{'학번': 1000, '이름': '홍길동', '연락처': '010-1111-2222'}
```

37/59

Section05 딕셔너리

- 동일한 키를 갖는 딕셔너리를 생성하는 것이 아니라 마지막에 있는 키가 적용

```
student1 = {'학번': 1000, '이름': '홍길동', '학과': '파이썬학과', '학번': 2000}
student1
```

출력 결과

```
{'학번': 2000, '이름': '홍길동', '학과': '파이썬학과'}
```

38/59

Section05 딕셔너리

- **딕셔너리의 사용**
  - 키로 값에 접근하는 코드

```
student1['학번']
student1['이름']
student1['학과']
```

출력 결과

```
2000
'홍길동'
'파이썬학과'
```

- 딕셔너리명.get(키) 함수를 사용해 키로 값에 접근

```
student1.get('이름')
```

출력 결과

```
'홍길동'
```

39/59

Section05 딕셔너리

- 딕셔너리명[키]와 딕셔너리명.get(키)는 결과 같음
- 딕셔너리명[키]는 없는 키 호출하면 오류 나지만 딕셔너리명.get(키)는 없는 키를 호출하면 아무것도 반환하지 않음
- 없는 키를 찾을 때 딕셔너리명.get(키)를 사용

```
student1['주소']
student1.get('주소')
```

- 딕셔너리명.keys()는 딕셔너리의 모든 키 반환

```
student1.keys()
```

출력 결과

```
dict_keys(['학번', '이름', '학과'])
```

40/59

Section05 딕셔너리

- 출력 결과의 dict_keys가 보기 싫으면 list(딕셔너리명.keys()) 함수 사용

```
list(student1.keys())
```

출력 결과

```
['학번', '이름', '학과']
```

- 딕셔너리명.values() 함수는 딕셔너리의 모든 값을 리스트로 만들어 반환
- 딕셔너리명.values() 함수도 출력 결과의 dict_values가 보기 싫으면 list(딕셔너리명.values()) 함수 사용

```
student1.values()
```

출력 결과

```
dict_values([2000, '홍길동', '파이썬학과'])
```


Section05 딕셔너리

- 딕셔너리명.items() 함수를 사용하면 튜플 형태로도 구할 수 있음

```
student1.items()
```

출력 결과

```
dict_items([('학번', 2000), ('이름', '홍길동'), ('학과', '파이썬학과')])
```

- 딕셔너리 안에 해당 키가 있는지 없는지는 in을 사용해 확인
- 딕셔너리에 키가 있다면 True를 반환하고, 없다면 False를 반환

```
'이름' in student1
'주소' in student1
```

출력 결과

```
True
False
```


Section05 딕셔너리

- for 문을 활용해 딕셔너리의 모든 값을 출력하는 코드

Code07-08.py

```
1 singer = {}
2
3 singer['이름'] = '트와이스'
4 singer['구성원 수'] = 9
5 singer['데뷔'] = '서바이벌 식스틴'
6 singer['대표곡'] = 'SIGNAL'
7
8 for k in singer.keys() :
9     print('%s --> %s' % (k, singer[k]))
```

1행 : 빈 딕셔너리를 준비

3~6행 : 쌍을 만들어 딕셔너리에 추가

8~9행 : 모든 키와 값을 출력

출력 결과

```
이름 --> 트와이스
구성원 수 --> 9
데뷔 --> 서바이벌 식스틴
대표곡 --> SIGNAL
```


Section05 딕셔너리

- **딕셔너리의 정렬**
  - 키로 정렬한 후 딕셔너리 추출

Code07-09.py

```
1 import operator
2
3 trainDic, trainList = {}, []
4
5 trainDic = {'Thomas':'토마스', 'Edward':'에드워드', 'Henry':'헨리', 'Gothen':'고든',
              'James':'제임스'}
6 trainList = sorted(trainDic.items(), key = operator.itemgetter(0))
7
8 print(trainList)
```

6행의 operator.itemgetter() 함수를 사용하려고 1행에서 operator를 임포트

3행 : 빈 딕셔너리와 리스트를 준비
5행 : 딕셔너리 작성
6행 : 키를 기준으로 딕셔너리 정렬

출력 결과

```
[('Edward', '에드워드'), ('Gothen', '고든'), ('Henry', '헨리'), ('James', '제임스'),
('Thomas', '토마스')]
```

Section06 리스트, 튜플, 딕셔너리의 심화 내용

- **세트**
  - 키만 모아 놓은 딕셔너리의 특수한 형태
  - 딕셔너리의 키는 중복되면 안 되므로 세트에 들어 있는 값은 항상 유일
  - 세트를 생성하려면 딕셔너리처럼 중괄호 { } 사용하지만 : 없이 값을 입력
  - 중복된 키는 자동으로 하나만 남음

```
mySet1 = {1, 2, 3, 3, 3, 4}
mySet1
```

출력 결과

```
{1, 2, 3, 4}
```

  - 판매된 물품의 전체 수량이 아닌 종류만 파악하고 싶을 때

```
salesList = ['삼각김밥', '바나나', '도시락', '삼각김밥', '삼각김밥', '도시락', '삼각김밥']
set(salesList)
```

출력 결과

```
{'도시락', '바나나', '삼각김밥'}
```


Section06 리스트, 튜플, 딕셔너리의 심화 내용

- 두 세트 사이의 교집합, 합집합, 차집합, 대칭 차집합을 구할 때

```
mySet1 = {1, 2, 3, 4, 5}
mySet2 = {4, 5, 6, 7}
mySet1 & mySet2     # 교집합
mySet1 | mySet2     # 합집합
mySet1 - mySet2     # 차집합
mySet1 ^ mySet2     # 대칭 차집합
```

출력 결과

```
{4, 5}
{1, 2, 3, 4, 5, 6, 7}
{1, 2, 3}
{1, 2, 3, 6, 7}
```

- 연산자 &, |, -, ^ 대신 함수를 사용

```
mySet1.intersection(mySet2)           # 교집합
mySet1.union(mySet2)                  # 합집합
mySet1.difference(mySet2)             # 차집합
mySet1.symmetric_difference(mySet2)   # 대칭 차집합
```


Section06 리스트, 튜플, 딕셔너리의 심화 내용

- **컴프리헨션**
  - 값이 순차적인 리스트를 한 줄로 만드는 간단한 방법
  - 1부터 5까지 저장된 리스트

```
numList = []
for num in range(1, 6) :
    numList.append(num)
numList
```

출력 결과

```
[1, 2, 3, 4, 5]
```

  - 컴프리헨션으로 작성

```
numList = [num for num in range(1, 6)]
numList
```

출력 결과

```
[1, 2, 3, 4, 5]
```


Section06 리스트, 튜플, 딕셔너리의 심화 내용

- 컴프리헨션의 구성

```
리스트 = [수식 for 항목 in range() if 조건식]
```

- 1~5의 제곱으로 구성된 리스트

```
numList = [num * num for num in range(1, 6)]
numList
```

출력 결과

```
[1, 4, 9, 16, 25]
```

- 1~20 숫자 중에서 3의 배수로만 리스트를 구성

```
numList = [num for num in range(1, 21) if num % 3 == 0]
numList
```

출력 결과

```
[3, 6, 9, 12, 15, 18]
```

Section06 리스트, 튜플, 딕셔너리의 심화 내용

- **동시에 여러 리스트에 접근**
  - zip() 함수를 사용해 동시에 여러 리스트에 접근

```
foods = ['떡볶이', '짜장면', '라면', '피자', '맥주', '치킨', '삼겹살']
sides = ['오뎅', '단무지', '김치']
for food, side in zip(foods, sides) :
    print(food, ' --> ', side)
```

출력 결과

```
떡볶이 --> 오뎅
짜장면 --> 단무지
라면 --> 김치
```


Section06 리스트, 튜플, 딕셔너리의 심화 내용

- 두 리스트를 튜플이나 딕셔너리로 짝지을 때 zip() 함수 사용

```
foods = ['떡볶이', '짜장면', '라면', '피자', '맥주', '치킨', '삼겹살']
sides = ['오뎅', '단무지', '김치']
tupList = list(zip(foods, sides))
dic = dict(zip(foods, sides))
tupList
dic
```

출력 결과

```
[('떡볶이', '오뎅'), ('짜장면', '단무지'), ('라면', '김치')]
{'떡볶이': '오뎅', '짜장면': '단무지', '라면': '김치'}
```


Section06 리스트, 튜플, 딕셔너리의 심화 내용

- 리스트의 복사
  - 얕은 복사 : newList=oldList는 newList와 oldList가 동일한 메모리 공간 공유

```
oldList = ['짜장면', '탕수육', '군만두']
newList = oldList
print(newList)
oldList[0] = '짬뽕'
oldList.append('깐풍기')
print(newList)
```

출력 결과

```
['짜장면', '탕수육', '군만두']
['짬뽕', '탕수육', '군만두', '깐풍기']
```

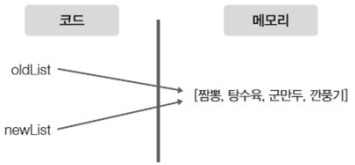


그림 7-7 newList=oldList를 이용한 리스트 복사

Section06 리스트, 튜플, 딕셔너리의 심화 내용

- 얕은 복사의 방지
  - 깊은 복사 : newList=oldList[:]는 메모리의 공간을 복사해서 새로 만듬

```
oldList = ['짜장면', '탕수육', '군만두']
newList = oldList[:]
print(newList)
oldList[0] = '짬뽕'
oldList.append('깐풍기')
print(newList)
```

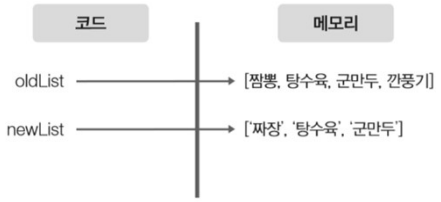


그림 7-8 newList=oldList[:]를 이용한 리스트 복사

Section06 리스트, 튜플, 딕셔너리의 심화 내용

- **리스트를 이용한 스택 구현**
  - 스택(Stack) : 한 쪽 끝이 막혀 먼저 들어간 것이 가장 나중에 나오는 형태의 자료구조

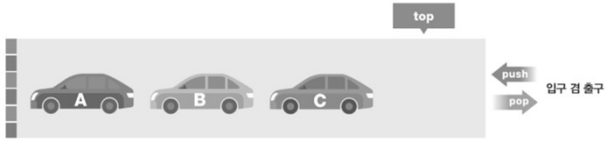


그림 7-9 **스택의 구조와 원리**

- LIFO(Last In First Out) 구조 : 가장 나중에 들어간 것이 가장 먼저 나오는 구조
- Top : 자동차 세 대 중 가장 마지막에 들어간 자동차 C 다음의 비어 있는 위치
  - top은 스택에 들어 있는 데이터 중 가장 마지막 데이터의 바로 다음 위치
- 푸시(Push) : 데이터를 넣는 것
- 팝(Pop) : 데이터를 빼는 것

Section06 리스트, 튜플, 딕셔너리의 심화 내용

- 한 쪽이 막힌 주차장을 만들어 리스트로 스택 구현

```
parking = []
top = 0
```

그림 7-10 **비어 있는 주차장**

Section06 리스트, 튜플, 딕셔너리의 심화 내용

- append() 함수 사용하여 자동차 한 대 넣어 보자(푸시)

```
parking.append('자동차A')
top += 1
```

그림 7-11 **주차장에 자동차 한 대 넣기**

Section06 리스트, 튜플, 딕셔너리의 심화 내용

- 자동차 B와 자동차 C도 주차장에 넣기

```
parking.append('자동차B')
top += 1
parking.append('자동차C')
top += 1
```

그림 7-12 **주차장에 자동차를 두 대 추가로 넣기**

Section06 리스트, 튜플, 딕셔너리의 심화 내용

- 자동차 빼기

```
top -= 1
outCar = parking.pop()
print(outCar)
```

그림 7-13 **주차장에서 자동차 한 대 빼기**

Section06 리스트, 튜플, 딕셔너리의 심화 내용

- 자동차 빼기

```
top -= 1
outCar = parking.pop()
print(outCar)
```

그림 7-13 **주차장에서 자동차 한 대 빼기**

Tip • 스택과 함께 많이 사용되는 자료구조 큐(Queue). 큐는 양쪽이 뚫린 구조. 파이프 같은 모양, 한 쪽으로 들어가면 다른 한 쪽으로 나오는 형태