Ch5. 딕셔너리 ,
리스트 튜플 딕셔너리 응용

- **딕셔너리 사용하기**
  - 파이썬에서는 연관된 값을 묶어서 저장하는 용도로 딕셔너리라는 자료형을 제공함
  - 게임 캐릭터의 능력치를 딕셔너리에 저장해보자

```
lux = {'health': 490, 'mana': 334, 'melee': 550, 'armor': 18.72}
```

  - 딕셔너리만 봐도 lux라는 캐릭터의 체력(health)은 490, 마나(mana)는 334, 사거리(melee)는 550, 방어력(armor)은 18.72라는 것을 쉽게 알 수 있음
  - 딕셔너리는 값마다 이름을 붙여서 저장하는 방식임
  - **사전(dictionary)에서 단어를 찾듯이 값을 가져올 수 있다고 하여 딕셔너리라고 부름**

- **딕셔너리 만들기**
  - 딕셔너리는 { }(중괄호) 안에 키: 값 형식으로 저장하며 각 키와 값은 ,(콤마)로 구분
    - 딕셔너리 = {키1: 값1, 키2: 값2}
  - 키와 값이 4개씩 들어있는 딕셔너리를 만들어보자

```
>>> lux = {'health': 490, 'mana': 334, 'melee': 550, 'armor': 18.72}
>>> lux
{'health': 490, 'mana': 334, 'melee': 550, 'armor': 18.72}
```

  - **딕셔너리는 키를 먼저 지정하고 :(콜론)을 붙여서 값을 표현함**
  - **키에는 값을 하나만 지정할 수 있으며 이런 특성을 따서 키-값 쌍(key-value pair)이라 부름(키-값은 1:1 대응)**

Key	value
Health	490
Mana	334
Melee	550
armor	18.72

- **키 이름이 중복되면?**
  - 딕셔너리를 만들 때 키 이름이 중복되면 어떻게 될까? (파이썬 3.6 기준)

```
>>> lux = {'health': 490, 'health': 800, 'mana': 334, 'melee': 550, 'armor': 18.72}
>>> lux['health']    # 키가 중복되면 가장 뒤에 있는 값만 사용함
800
>>> lux    # 중복되는 키는 저장되지 않음
{'health': 800, 'mana': 334, 'melee': 550, 'armor': 18.72}
```

  - 딕셔너리 lux를 만들 때 'health': 490이 있고 그 뒤에 'health': 800을 넣었음
  - 키 'health'가 중복됨
  - lux['health']를 출력해보면 800이 나옴
  - **딕셔너리에 키와 값을 저장할 때 키가 중복되면 가장 뒤에 있는 값만 사용**
  - **중복되는 키는 저장되지 않음**

- **딕셔너리 키의 자료형**
  - 딕셔너리의 키는 문자열뿐만 아니라 정수, 실수, 불도 사용할 수 있으며 자료형을 섞어서 사용해도 됨
  - 값에는 리스트, 딕셔너리 등을 포함하여 모든 자료형을 사용할 수 있음

```
>>> x = {100: 'hundred', False: 0, 3.5: [3.5, 3.5]}
>>> x
{100: 'hundred', False: 0, 3.5: [3.5, 3.5]}
```

  - **key에는 리스트와 딕셔너리를 사용할 수 없음**

```
>>> x = {[10, 20]: 100}
Traceback (most recent call last):
  File "<pyshell#3>", line 1, in <module>
    x = {[10, 20]: 100}
TypeError: unhashable type: 'list'
>>> x = {{'a': 10}: 100}
Traceback (most recent call last):
  File "<pyshell#4>", line 1, in <module>
    x = {{'a': 10}: 100}
TypeError: unhashable type: 'dict'
```

- ## 빈 딕셔너리 만들기
  - 딕셔너리 = {}
  - 딕셔너리 = dict()

```
>>> x = {}
>>> x
{}
>>> y = dict()
>>> y
{}
```

- **dict로 딕셔너리 만들기**
  - dict는 다음과 같이 키와 값을 연결하거나, 리스트, 튜플, 딕셔너리로 딕셔너리를 만들 때 사용함
    - 딕셔너리 = dict(키1=값1, 키2=값2)
    - 딕셔너리 = dict(zip([키1, 키2], [값1, 값2]))
    - 딕셔너리 = dict([(키1, 값1), (키2, 값2)])
    - 딕셔너리 = dict({키1: 값1, 키2: 값2})
  - **dict에서 키=값 형식으로 딕셔너리를 만들 수 있음**
  - **키에 ' '(작은따옴표)나 " "(큰따옴표)를 사용하지 않아야 함**
  - **키는 딕셔너리를 만들고 나면 문자열로 바뀜**

```
>>> lux1 = dict(health=490, mana=334, melee=550, armor=18.72)    # 키=값 형식으로 딕셔너리를 만듦
>>> lux1
{'health': 490, 'mana': 334, 'melee': 550, 'armor': 18.72}
```

- **dict로 딕셔너리 만들기**
  - **2 : dict에서 zip 함수를 이용하는 방법**
  - 키가 들어있는 리스트와 값이 들어있는 리스트를 차례대로 zip에 넣은 뒤 다시 dict에 넣어주면 됨

```
>>> lux2 = dict(zip(['health', 'mana', 'melee', 'armor'], [490, 334, 550, 18.72]))    # zip 함수로
>>> lux2                                                                  # 키 리스트와 값 리스트를 묶음
{'health': 490, 'mana': 334, 'melee': 550, 'armor': 18.72}
```

  - 키와 값을 리스트가 아닌 튜플에 저장해서 zip에 넣어도 됨
  - **3 : 리스트 안에 (키, 값) 형식의 튜플을 나열하는 방법**

```
>>> lux3 = dict([('health', 490), ('mana', 334), ('melee', 550), ('armor', 18.72)])
>>> lux3                                                  # (키, 값) 형식의 튜플로 딕셔너리를 만듦
{'health': 490, 'mana': 334, 'melee': 550, 'armor': 18.72}
```

- **dict로 딕셔너리 만들기**
  - **4 : dict 안에서 중괄호로 딕셔너리를 생성하는 방법**

```
>>> lux4 = dict({'health': 490, 'mana': 334, 'melee': 550, 'armor': 18.72})     # dict 안에서
>>> lux4                                                                        # 중괄호로 딕셔너리를 만듦
{'health': 490, 'mana': 334, 'melee': 550, 'armor': 18.72}
```

  - 딕셔너리는 키를 통해서 값의 의미를 파악하기 쉬움
  - 딕셔너리는 예제의 게임 캐릭터 능력치처럼 특정 주제에 대해 연관된 값들을 모아둘 때 주로 사용

- **cf) 내장함수 zip**
  - zip은 동일한 개수로 이루어진 자료형을 묶어 주는 역할을 하는 함수

```
>>> list(zip([1,2,3],[4,5,6]))
[(1, 4), (2, 5), (3, 6)]
>>> list(zip([1,2,3],[4,5,6],[7,8,9]))
[(1, 4, 7), (2, 5, 8), (3, 6, 9)]
>>> list(zip("abc","def"))
[('a', 'd'), ('b', 'e'), ('c', 'f')]
>>>
```

  - 같은 길이의 리스트를 같은 인덱스끼리 묶어 반환
  - 여럿의 리스트를 짝지어 줌
  - dict()의 경우 key, value 로 구성되어 있으므로 3개이상은 묶을 수 없음

```
a="test"
b=[1,2,3]
c=("하나","둘","셋")
print(list(zip(a,b,c)))
print(dict(zip(a,b)))
```

```
[('t', 1, '하나'), ('e', 2, '둘'), ('s', 3, '셋')]
{'t': 1, 'e': 2, 's': 3}
>>>
```

```
a="test"
b=[1,2,3]
c=("하나","둘","셋")
print(dict(zip(a,b,c)))
```

```
Traceback (most recent call last):
  File "D:/private/python/test0403_dic.py", line 4, in <module>
    print(dict(zip(a,b,c)))
ValueError: dictionary update sequence element #0 has length 3; 2 is required
```

- **딕셔너리의 키에 접근하고 값 할당하기**
  - 딕셔너리의 키에 접근할 때는 딕셔너리 뒤에 [ ](대괄호)를 사용하며 [ ] 안에 키를 지정해주면 됨

• 딕셔너리[키]

```
>>> lux = {'health': 490, 'mana': 334, 'melee': 550, 'armor': 18.72}
>>> lux['health']
490
>>> lux['armor']
18.72
```

- **딕셔너리의 키에 값 할당하기**
  - **딕셔너리는 [ ]로 키에 접근한 뒤 값을 할당함**

• 딕셔너리[키] = 값

```
>>> lux = {'health': 490, 'mana': 334, 'melee': 550, 'armor': 18.72}
>>> lux['health'] = 2037    # 키 'health'의 값을 2037로 변경
>>> lux['mana'] = 1184      # 키 'mana'의 값을 1184로 변경
>>> lux
{'health': 2037, 'mana': 1184, 'melee': 550, 'armor': 18.72}
```

  - 딕셔너리에서 키의 값을 출력할 때와 마찬가지로 [ ]에 키를 지정한 뒤 값을 할당하면 됨
  - 딕셔너리에 없는 키에 값을 할당하면 해당 키가 추가되고 값이 할당됨

```
>>> lux['mana_regen'] = 3.28    # 키 'mana_regen'을 추가하고 값 3.28 할당
>>> lux
{'health': 2037, 'mana': 1184, 'melee': 550, 'armor': 18.72, 'mana_regen': 3.28}
```

- **딕셔너리의 키에 값 할당하기**
  - 딕셔너리에 없는 키에서 값을 가져오려고 하면 에러가 발생

```
>>> lux = {'health': 490, 'mana': 334, 'melee': 550, 'armor': 18.72}
>>> lux['attack_speed']    # lux에는 'attack_speed' 키가 없음
Traceback (most recent call last):
  File "<pyshell#3>", line 1, in <module>
    lux['attack_speed']
KeyError: 'attack_speed'
```

- **딕셔너리의 키에 값 삭제하기**
  - del 함수를 이용하여 삭제
  - **del lux[key]**

```
>>> lux={'health' : 490, 'mana' : 334, 'melee' : 550, 'armor' : 18.72 }
>>> lux
{'health': 490, 'mana': 334, 'melee': 550, 'armor': 18.72}
>>> del lux['health']
>>> lux
{'mana': 334, 'melee': 550, 'armor': 18.72}
```

- **딕셔너리에 키가 있는지 확인하기**
  - 딕셔너리에서 **키가 있는지 확인하고 싶다면 in 연산자를 사용**하면 됨
    - **키 in 딕셔너리**

```
>>> lux = {'health': 490, 'mana': 334, 'melee': 550, 'armor': 18.72}
>>> 'health' in lux
True
>>> 'attack_speed' in lux
False
```

  - 딕셔너리에 특정 키가 있으면 True, 없으면 False가 나옴
  - 딕셔너리 lux에 키 'health'가 있으므로 True, 'attack_speed'가 없으므로 False
  - 반대로 in 앞에 not을 붙이면 특정 키가 없는지 확인함
    - **키 not in 딕셔너리**

```
>>> 'attack_speed' not in lux
True
>>> 'health' not in lux
False
```

  - **not in**은 특정 키가 없으면 True, 있으면 False가 나옴

- **딕셔너리에 키가 있는지 확인하기**
  - 딕셔너리를 사용하다 보면 딕셔너리의 키 개수(길이)를 구할 필요가 있음
  - 딕셔너리의 키와 값을 직접 타이핑할 때는 키의 개수를 알기가 쉬움
  - 실무에서는 함수 등을 사용해서 딕셔너리를 생성하거나 키를 추가하기 때문에 키의 개수가 눈에 보이지 않음
  - **키의 개수는 len 함수를 사용하여 구함**
    (키와 값은 1:1 관계이므로 키의 개수는 곧 값의 개수임)
  - len(lux)와 같이 len에 딕셔너리 변수를 넣어서 키의 개수를 구해도 되고, len에 딕셔너리를 그대로 넣어도 됨

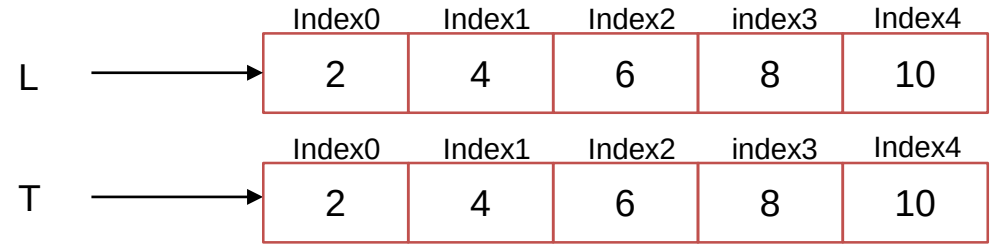
• len(딕셔너리)

```
>>> lux = {'health': 490, 'mana': 334, 'melee': 550, 'armor': 18.72}
>>> len(lux)
4
>>> len({'health': 490, 'mana': 334, 'melee': 550, 'armor': 18.72})
4
```

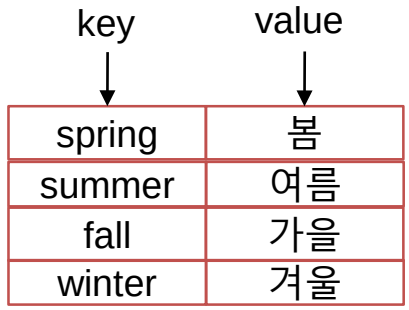
- 딕셔너리 정리
  - 딕셔너리를 생성할 때는 { }(중괄호)를 사용하고, 키와 값을 1:1 관계로 저장한다는 점이 중요
  - 딕셔너리는 특정 주제에 대해 연관된 값을 저장할 때 사용한다는 점 기억
  - 이 부분이 리스트, 튜플과 딕셔너리의 차이점

- 딕셔너리와 리스트 튜플의 구조 비교

```
>>> L = [2, 4, 6, 8, 10] #list
>>> T = (2, 4, 6, 8, 10) #tuple
```




```
>>> D = {'spring':'봄', 'summer':'여름', 'fall':'가을', 'winter':'겨울'}
```



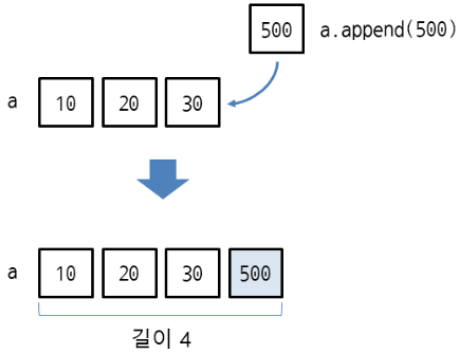
- **리스트와 튜플 응용**
  - 파이썬의 리스트는 생각보다 기능이 많은데, 요소를 추가/삭제하거나, 정보를 조회하는 메서드(함수)도 제공함
  - for 반복문과 결합하면 연속적이고 반복되는 값을 손쉽게 처리할 수 있음

- **리스트 조작하기**
  - 리스트를 조작하는 메서드(method)임(메서드는 객체에 속한 함수를 뜻함)

- **리스트에 요소 추가하기**
  - append: 요소 하나를 추가
  - extend: 리스트를 연결하여 확장
  - insert: 특정 인덱스에 요소 추가

- **리스트에 요소 하나 추가하기**
  - 리스트 [10, 20, 30]에 500을 추가하여 리스트는 [10, 20, 30, 500]이 됨
    (메서드를 호출한 리스트가 변경되며 새 리스트는 생성되지 않음)

```
>>> a = [10, 20, 30]
>>> a.append(500)
>>> a
[10, 20, 30, 500]
>>> len(a)
4
```



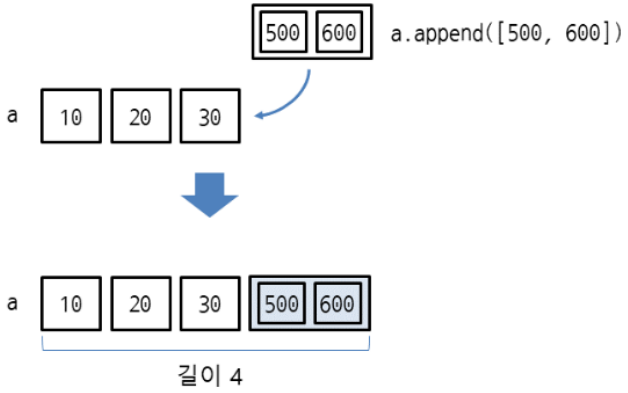
- **빈 리스트에 요소 하나 추가하기**

```
>>> a = []
>>> a.append(10)
>>> a
[10]
```

- **리스트 안에 리스트 추가하기**
  - append는 append(리스트)처럼 리스트를 넣으면 중첩리스트 가능함

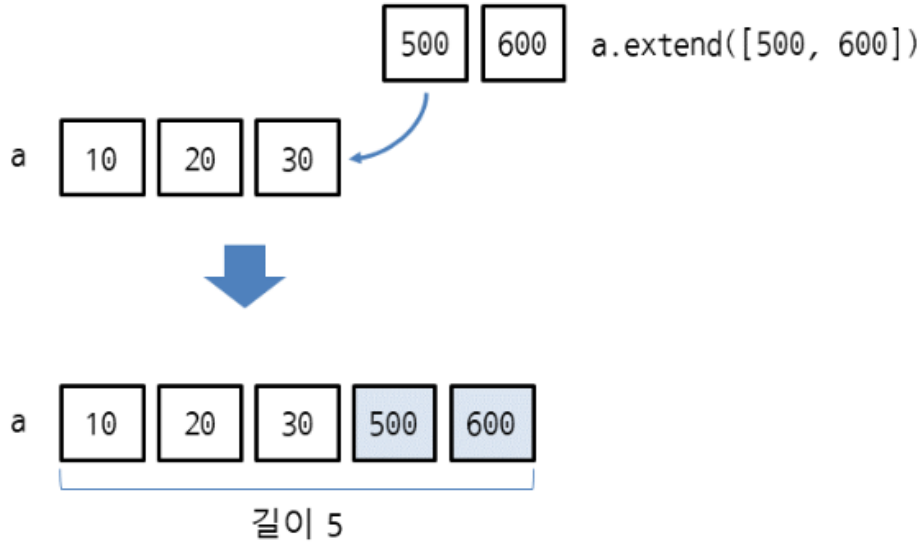
```
>>> a = [10, 20, 30]
>>> a.append([500, 600])
>>> a
[10, 20, 30, [500, 600]]
>>> len(a)
4
```

Append : 항상 길이가 1씩 증가



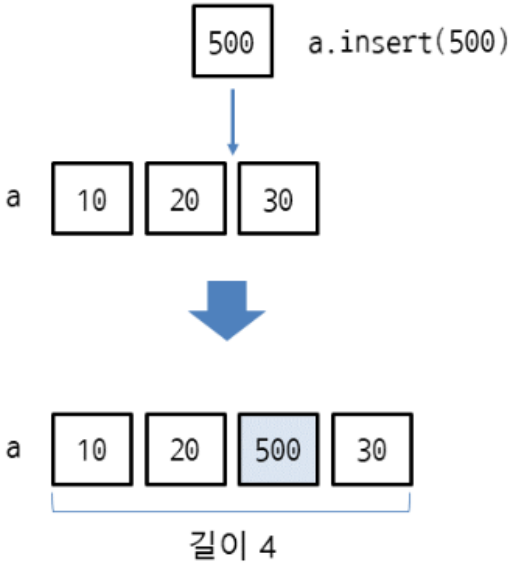
- **리스트 확장하기**
  - **extend(리스트) : 리스트 끝에 다른 리스트를 연결하여 리스트를 확장**함

```
>>> a = [10, 20, 30]
>>> a.extend([500, 600])
>>> a
[10, 20, 30, 500, 600]
>>> len(a)
5
```



- 리스트의 특정 인덱스에 요소 추가하기
  - 원하는 위치에 추가하고 싶은 경우 사용
  - insert(인덱스, 요소) : 리스트의 특정 인덱스에 요소 하나를 추가함

```
>>> a = [10, 20, 30]
>>> a.insert(2, 500)
>>> a
[10, 20, 500, 30]
>>> len(a)
4
```



- **리스트의 특정 인덱스에 요소 추가하기**
  - **insert(0, 요소)**: 리스트의 맨 처음에 요소를 추가
  - **insert(len(리스트), 요소)**: 리스트 끝에 요소를 추가

```
>>> a = [10, 20, 30]
>>> a.insert(0, 500)
>>> a
[500, 10, 20, 30]
```

  - insert에 **마지막 인덱스보다 큰 값을 지정하면 리스트 끝에 요소 하나를 추가**

```
>>> a = [10, 20, 30]
>>> a.insert(len(a), 500)
>>> a
[10, 20, 30, 500]
```

- **리스트의 특정 인덱스에 요소 추가하기**
  - len(리스트)는 마지막 인덱스보다 1이 더 크기 때문에 리스트 끝에 값을 추가할 때 자주 활용
  - insert는 요소 하나를 추가하므로 insert에 리스트를 넣으면 append처럼 리스트 안에 리스트가 들어감

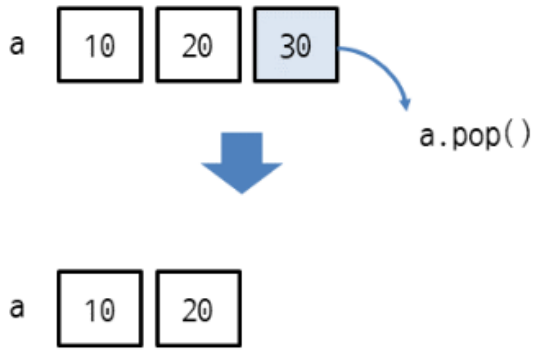
```
>>> a = [10, 20, 30]
>>> a.insert(1, [500, 600])
>>> a
[10, [500, 600], 20, 30]
```

```
>>> a = [10, 20, 30]
>>> a[1:1] = [500, 600]
>>> a
[10, 500, 600, 20, 30]
```

- **리스트에서 요소 삭제하기**
  - pop: 마지막 요소 또는 특정 인덱스의 요소를 삭제
  - remove: 특정 값을 찾아서 삭제

- **리스트에서 특정 인덱스의 요소를 삭제하기**
  - **pop() : 리스트의 마지막 요소를 삭제한 뒤 삭제한 요소를 반환함**

```
>>> a = [10, 20, 30]
>>> a.pop()
30
>>> a
[10, 20]
```



- 리스트에서 특정 인덱스의 요소를 삭제하기
  - pop(인덱스) : 해당 인덱스의 요소를 삭제한 뒤 삭제한 요소를 반환

```
>>> a = [10, 20, 30]
>>> a.pop(1)
20
>>> a
[10, 30]
```

  - pop 대신 del을 사용해도 상관없음

```
>>> a = [10, 20, 30]
>>> del a[1]
>>> a
[10, 30]
```

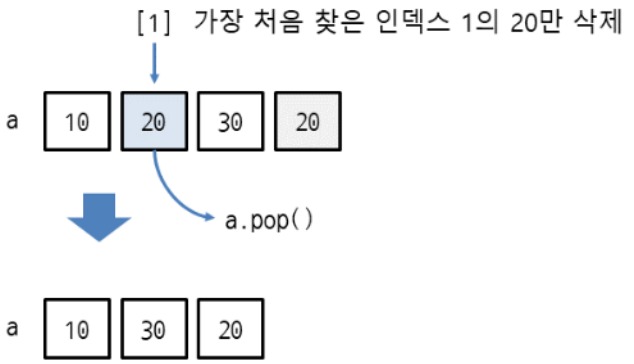
- **리스트에서 특정 값을 찾아서 삭제하기**
  - **remove(값) : 리스트에서 특정 값을 찾아서 삭제**

```
>>> a = [10, 20, 30]
>>> a.remove(20)
>>> a
[10, 30]
```

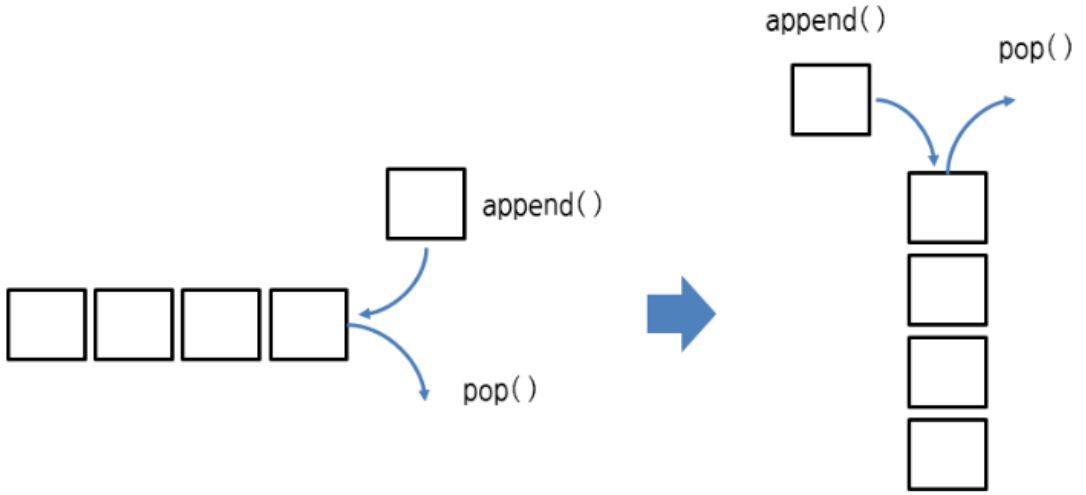
  - **리스트에 같은 값이 여러 개 있을 경우 처음 찾은 값을 삭제**

```
>>> a = [10, 20, 30, 20]
>>> a.remove(20)
>>> a
[10, 30, 20]
```

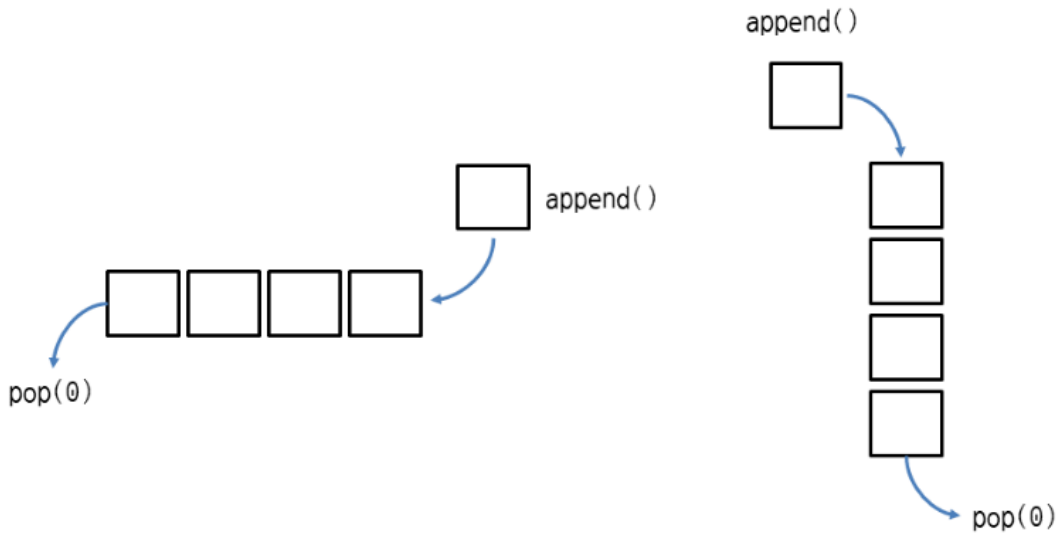
  - 리스트 a에 20이 2개 있지만
    가장 처음 찾은 인덱스 1의 20만 삭제



- **cf) 리스트로 스택과 큐 만들기**



stack



queue

- **cf) deque (반복 가능한 개체)**

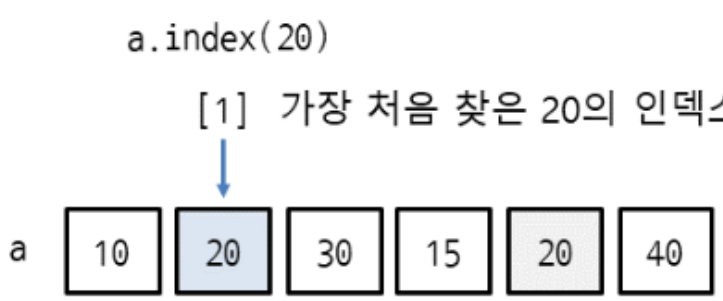
```
>>> from collections import deque    # collections 모듈에서 deque를 불러옴
>>> a=deque([10,20,30])
>>> a
deque([10, 20, 30])
>>> a.append(500)  # 오른쪽에 500추가
>>> a
deque([10, 20, 30, 500])
>>> a.popleft()       # 왼쪽 요소 삭제
10
>>> a
deque([20, 30, 500])
>>>

deque([20, 30, 500])
>>> a.appendleft(600) # 왼쪽에 600추가
>>> a
deque([600, 20, 30, 500])
>>> a.pop()              # 오른쪽 요소 삭제
500
>>> a
deque([600, 20, 30])
```

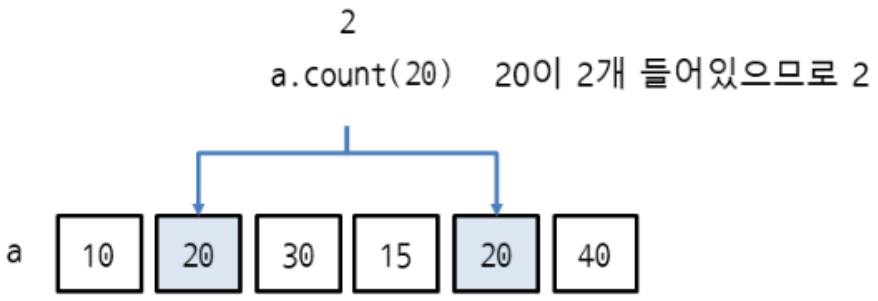
- **리스트에서 특정 값의 인덱스 구하기**
  - **index(값) : 리스트에서 특정 값의 인덱스를 구함**
  - 같은 값이 여러 개일 경우 처음 찾은 인덱스를 구함

```
>>> a = [10, 20, 30, 15, 20, 40]
>>> a.index(20)
1
```

- **특정 값의 개수 구하기**
  - **count(값) : 리스트에서 특정 값의 개수를 구함**

```
>>> a = [10, 20, 30, 15, 20, 40]
>>> a.count(20)
2
```





- **리스트의 순서를 뒤집기**
  - **reverse() : 리스트에서 요소의 순서를 반대로 뒤집음**

```
>>> a = [10, 20, 30, 15, 20, 40]
>>> a.reverse()
>>> a
[40, 20, 15, 30, 20, 10]
```

- **리스트의 요소를 정렬하기**
  - **sort() : 리스트의 요소을 작은 순서대로 정렬함(오름차순)**

- sort() 또는 sort(reverse=False): 리스트의 값을 작은 순서대로 정렬(오름차순)
- sort(reverse=True): 리스트의 값을 큰 순서대로 정렬(내림차순)

```
>>> a = [10, 20, 30, 15, 20, 40]
>>> a.sort()
>>> a
[10, 15, 20, 20, 30, 40]
```

```
>>> a.sort(reverse=True)
>>> a
[40, 30, 20, 20, 15, 10]
```

- **리스트의 모든 요소를 삭제하기**
  - **clear() : 리스트의 모든 요소를 삭제함**

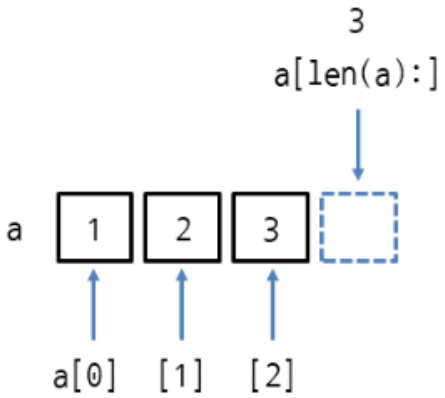
```
>>> a = [10, 20, 30]
>>> a.clear()
>>> a
[]
```

  - **clear 대신 del a[:]**와 같이 시작, 끝 인덱스를 생략하여 리스트의 모든 요소를 삭제할 수도 있음

```
>>> a = [10, 20, 30]
>>> del a[:]
>>> a
[]
```

- **리스트를 슬라이스로 조작하기**
  - 리스트 끝에 값이 한 개 들어있는 리스트를 추가함

```
>>> a = [10, 20, 30]
>>> a[len(a):] = [500]
>>> a
[10, 20, 30, 500]
```



  - a[len(a):]는 시작 인덱스를 len(a)로 지정해서 리스트의 마지막 인덱스보다 1이 더 큰 상태임
  - 그림과 같이 리스트 끝에서부터 시작하겠다는 뜻임
    (이때는 리스트의 범위를 벗어난 인덱스를 사용할 수 있음)
  - a[len(a):] = [500]과 같이 값이 한 개 들어있는 리스트를 할당하면 리스트 a 끝에 값을 한 개 추가하며 a.append(500)과 같음

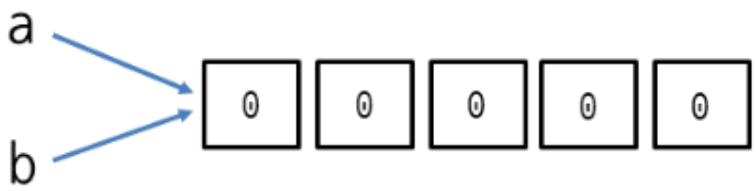
- **리스트를 슬라이스로 조작하기**
  - a[len(a):] = [500, 600]과 같이 요소가 여러 개 들어있는 리스트를 할당하면 리스트 a 끝에 다른 리스트를 연결하며 a.extend([500, 600])과 같음

```
>>> a = [10, 20, 30]
>>> a[len(a):] = [500, 600]
>>> a
[10, 20, 30, 500, 600]
```

- **리스트의 할당과 복사 알아보기**
  - 먼저 다음과 같이 리스트를 만든 뒤 다른 변수에 할당함

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

  - b = a와 같이 리스트를 다른 변수에 할당하면 리스트는 두 개가 될 것 같지만 실제로는 리스트가 한 개임

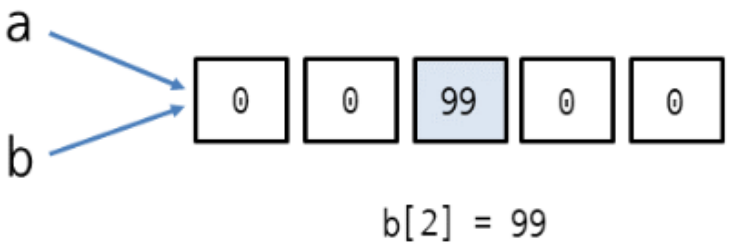


- **리스트의 할당과 복사 알아보기**
  - 변수 이름만 다를 뿐 리스트 a와 b는 같은 객체임

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

  - a와 b는 같으므로 b[2] = 99와 같이 리스트 b의 요소를 변경하면 리스트 a와 b에 모두 반영됨

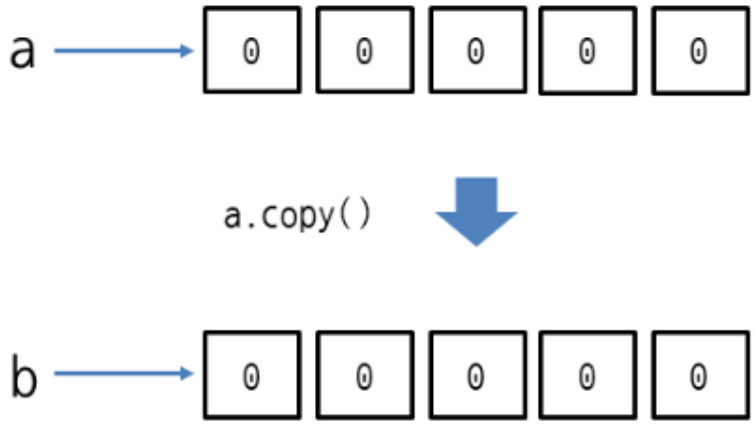
```
>>> b[2] = 99
>>> a
[0, 0, 99, 0, 0]
>>> b
[0, 0, 99, 0, 0]
```



- **리스트의 할당과 복사 알아보기**
  - **리스트 a와 b를 완전히 두 개로 만들려면 copy 메서드로 모든 요소를 복사**

```
>>> a = [0, 0, 0, 0, 0]
>>> b = a.copy()
```

  - **b = a.copy()**와 같이 copy를 사용한 뒤 b에 할당해주면
    리스트 a의 요소가 모두 b에 복사됨

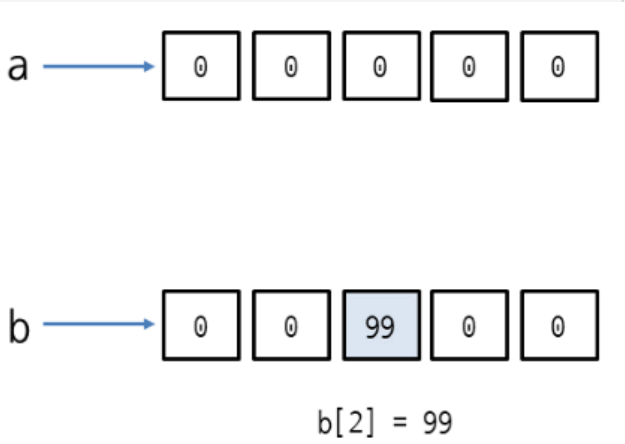


- **리스트의 할당과 복사 알아보기**
  - **a와 b를 is 연산자로 비교해보면 False가 나온 두 리스트는 다른 객체임**
  - **복사된 요소는 모두 같으므로 ==로 비교하면 True가 나옴**

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

  - 리스트 a와 b는 별개이므로 한쪽의 값을 변경해도 다른 리스트에 영향을 미치지 않음

```
>>> b[2] = 99
>>> a
[0, 0, 0, 0, 0]
>>> b
[0, 0, 99, 0, 0]
```



- **Cf) shallow copy, deep copy**
- **Shallow copy**
  - list의 슬라이싱을 통한 새로운 값을 할당
  - 슬라이싱을 통해서 값을 할당하면 새로운 id가 부여되며, 서로 영향을 X

```
>>> a = [1,2,3]
>>> b = a[:]
>>> id(a)
4396179528
>>> id(b)
4393788808
>>> a == b
True
>>> a is b
False
>>> b[0] = 5
>>> a
[1, 2, 3]
>>> b
[5, 2, 3]
```

```
>>> a = [[1,2], [3,4]]
>>> b = a[:]
>>> id(a)
4395624328
>>> id(b)
4396179592
>>> id(a[0])
4396116040
>>> id(b[0])
4396116040
```

- **Cf) shallow copy, deep copy (cont')**
  - 재할당의 경우는 문제가 없음, 메모리 주소도 변경
  - a[1]값이 변경하면 b1도 변경

```
>>> a[0] = [8,9]
>>> a
[[8, 9], [3, 4]]
>>> b
[[1, 2], [3, 4]]
>>> id(a[0])
4393788808
>>> id(b[0])
4396116040
```

```
>>> a[1].append(5)
>>> a
[[8, 9], [3, 4, 5]]
>>> b
[[1, 2], [3, 4, 5]]
>>> id(a[1])
4396389896
>>> id(b[1])
4396389896
```

```
>>> import copy
>>> a = [[1,2],[3,4]]
>>> b = copy.copy(a)
>>> a[1].append(5)
>>> a
[[1, 2], [3, 4, 5]]
>>> b
[[1, 2], [3, 4, 5]]
```

  - copy 모듈의 copy 메소드 또한 얕은 복사

- **Cf) shallow copy, deep copy**
- **deep copy**
  - 깊은 복사는 내부에 객체들까지 새롭게 copy
  - **copy.deepcopy메소드**

```
>>> import copy
>>> a = [[1,2],[3,4]]
>>> b = copy.deepcopy(a)
>>> a[1].append(5)
>>> a
[[1, 2], [3, 4, 5]]
>>> b
[[1, 2], [3, 4]]
```

- **가장 작은 수와 가장 큰 수 구하기**
  - 리스트를 작은 순서대로 정렬(오름차순)하면 첫 번째 요소가 가장 작은 수임
  - 반대로 큰 순서대로 정렬(내림차순)하면 첫 번째 요소가 가장 큰 수가 됨

```
>>> a = [38, 21, 53, 62, 19]
>>> a.sort()
>>> a[0]
19
>>> a.sort(reverse=True)
>>> a[0]
62
```

  - **간단한 방법은 파이썬에서 제공하는 min, max 함수를 사용하면 됨**

```
>>> a = [38, 21, 53, 62, 19]
>>> min(a)
19
>>> max(a)
62
```

  - min은 리스트에서 가장 작은 값을 구하고, max는 가장 큰 값을 구함

- **input().split()과 map**
  - input().split()을 사용한 뒤에 변수 한 개에 저장해보면 리스트인지 확인할 수 있음

```
>>> a = input().split()
10 20 (입력)
>>> a
['10', '20']
```

  - map을 사용해서 정수로 변환

```
>>> a = map(int, input().split())
10 20 (입력)
>>> a
<map object at 0x03DFB0D0>
>>> list(a)
[10, 20]
```

- **input().split()과 map**
  - 이 리스트 [10, 20]을 변수 두 개에 저장하면 지금까지 사용한 a, b = map(int, input().split())와 같은 동작이 됨

```
>>> a, b = [10, 20]
>>> a
10
>>> b
20
```

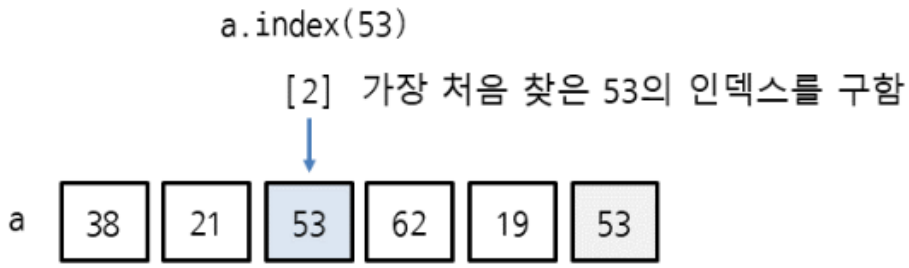
  - **map이 반환하는 맵 객체는 이터레이터라서 변수 여러 개에 저장하는 언패킹(unpacking)이 가능함**
  - a, b = map(int, input().split())처럼 list를 생략한 것임
  - a, b = map(int, input().split())을 풀어서 쓰면 다음과 같은 코드가 됨

```
x = input().split()     # input().split()의 결과는 문자열 리스트
m = map(int, x)         # 리스트의 요소를 int로 변환, 결과는 맵 객체
a, b = m                # 맵 객체는 변수 여러 개에 저장할 수 있음
```

- **튜플 응용하기**
  - 튜플은 리스트와는 달리 내용을 변경할 수 없음(불변, immutable)
  - 내용을 변경하는 append 같은 메서드는 사용할 수 없고,
    요소의 정보를 구하는 메서드만 사용할 수 있음

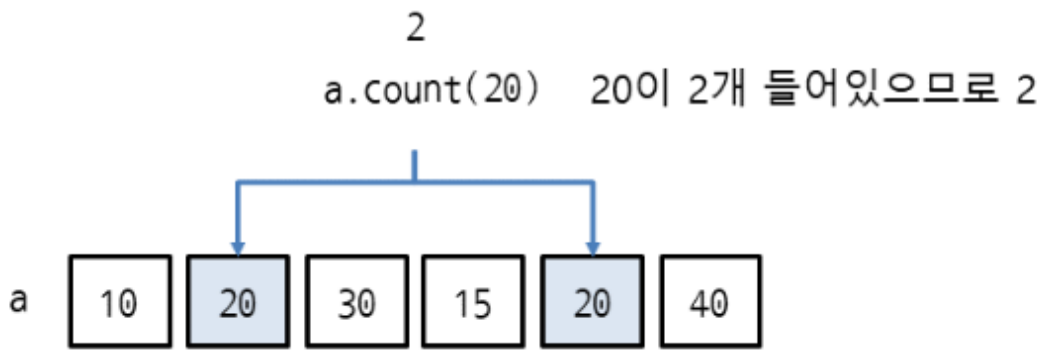
- **튜플에서 특정 값의 인덱스 구하기**
  - 같은 값이 여러 개일 경우 처음 찾은 인덱스를 구함(가장 작은 인덱스)

```
>>> a = (38, 21, 53, 62, 19, 53)
>>> a.index(53)
2
```



- **튜플에서 특정 값의 개수 구하기**

```
>>> a = (10, 20, 30, 15, 20, 40)
>>> a.count(20)
2
```



- **tuple에 map 사용하기**

```
>>> a = (1.2, 2.5, 3.7, 4.6)
>>> a = tuple(map(int, a))
>>> a
(1, 2, 3, 4)
```

- **튜플에서 가장 작은 수, 가장 큰 수, 합계 구하기**

```
>>> a = (38, 21, 53, 62, 19)
>>> min(a)
19
>>> max(a)
62
>>> sum(a)
193
```

- **딕셔너리에 키-값 쌍 추가하기**
  - 딕셔너리에 키-값 쌍을 추가하는 메서드는 두 가지가 있음

    - setdefault: 키-값 쌍 추가
    - update: 키의 값 수정, 키가 없으면 키-값 쌍 추가

  - **setdefault(키)는 딕셔너리에 키-값 쌍을 추가함**
  - setdefault에 키만 지정하면 값에 None을 저장함

```
>>> x = {'a': 10, 'b': 20, 'c': 30, 'd': 40}
>>> x.setdefault('e')
>>> x
{'a': 10, 'b': 20, 'c': 30, 'd': 40, 'e': None}
```

  - **setdefault(키, 기본값) : 값에 기본값을 저장한 뒤 해당 값을 반환함**

```
>>> x.setdefault('f', 100)
100
>>> x
{'a': 10, 'b': 20, 'c': 30, 'd': 40, 'e': None, 'f': 100}
```

- 딕셔너리에서 키의 값 수정하기
  - update(키=값) : 이름 그대로 딕셔너리에서 키의 값을 수정함

```
>>> x = {'a': 10, 'b': 20, 'c': 30, 'd': 40}
>>> x.update(a=90)
>>> x
{'a': 90, 'b': 20, 'c': 30, 'd': 40}
```

  - 딕셔너리에 키가 없으면 키-값 쌍을 추가함

```
>>> x.update(e=50)
>>> x
{'a': 90, 'b': 20, 'c': 30, 'd': 40, 'e': 50}
```

- 딕셔너리에서 키의 값 수정하기

```
>>> x.update(a=900, f=60)
>>> x
{'a': 900, 'b': 20, 'c': 30, 'd': 40, 'e': 50, 'f': 60}
```

  - update(키=값)은 키가 문자열일 때만 사용할 수 있음
  - 만약 키가 숫자일 경우에는 update(딕셔너리)처럼 딕셔너리를 넣어서 값을 수정할 수 있음

```
>>> y = {1: 'one', 2: 'two'}
>>> y.update({1: 'ONE', 3: 'THREE'})
>>> y
{1: 'ONE', 2: 'two', 3: 'THREE'}
```

- **딕셔너리에서 키의 값 수정하기**
  - **update(리스트), update(튜플)은 리스트와 튜플로 값을 수정함**
  - 리스트는 [[키1, 값1], [키2, 값2]] 형식으로 키와 값을 리스트로 만들고
    이 리스트를 다시 리스트 안에 넣어서 키-값 쌍을 나열해줌 (튜플도 같은 형식)

```
>>> y.update([[2, 'TWO'], [4, 'FOUR']])
>>> y
{1: 'ONE', 2: 'TWO', 3: 'THREE', 4: 'FOUR'}
```

  - 특히 update(반복가능한객체)는 키-값 쌍으로 된 반복 가능한 객체로 값을 수정함

```
>>> y.update(zip([1, 2], ['one', 'two']))
>>> y
{1: 'one', 2: 'two', 3: 'THREE', 4: 'FOUR'}
```

- **딕셔너리에서 키-값 쌍 삭제하기**
  - **pop(키) : 딕셔너리에서 특정 키-값 쌍을 삭제한 뒤 삭제한 값을 반환**
  - 다음은 딕셔너리 x에서 키 'a'를 삭제한 뒤 10을 반환함

```
>>> x = {'a': 10, 'b': 20, 'c': 30, 'd': 40}
>>> x.pop('a')
10
>>> x
{'b': 20, 'c': 30, 'd': 40}
```

  - pop(키, 기본값)처럼 기본값을 지정하면 딕셔너리에 키가 있을 때는
    해당 키-값 쌍을 삭제한 뒤 삭제한 값을 반환하지만 키가 없을 때는 기본값만 반환

```
>>> x.pop('z', 0)
0
```

- 딕셔너리에서 키-값 쌍 삭제하기
  - pop 대신 del로 특정 키-값 쌍을 삭제 : [ ]에 키를 지정하여 del을 사용

```
>>> x = {'a': 10, 'b': 20, 'c': 30, 'd': 40}
>>> del x['a']
>>> x
{'b': 20, 'c': 30, 'd': 40}
```

- **딕셔너리에서 임의의 키-값 쌍 삭제하기**
  - **popitem() : 딕셔너리에서 마지막 키-값 쌍을 삭제한 뒤 삭제한 키-값 쌍을 튜플로 반환함 (3.6ver)**

파이썬 3.6

```
>>> x = {'a': 10, 'b': 20, 'c': 30, 'd': 40}
>>> x.popitem()
('d', 40)
>>> x
{'a': 10, 'b': 20, 'c': 30}
```

  - **파이썬 3.5와 그 이하 버전에서** popitem 메서드를 사용하면 **임의의 키-값을 삭제**하므로 매번 삭제하는 키-값 쌍이 달라짐

파이썬 3.5

```
>>> x = {'a': 10, 'b': 20, 'c': 30, 'd': 40}
>>> x.popitem()    # 파이썬 3.5 이하에서는 매번 삭제하는 키-값 쌍이 달라짐
('a', 10)
>>> x
{'b': 20, 'c': 30, 'd': 40}
```

- **딕셔너리의 모든 키-값 쌍을 삭제하기**
  - **clear() : 딕셔너리의 모든 키-값 쌍을 삭제함**
  - 딕셔너리 x의 모든 키-값 쌍을 삭제하여 빈 딕셔너리 {}가 됨

```
>>> x = {'a': 10, 'b': 20, 'c': 30, 'd': 40}
>>> x.clear()
>>> x
{}
```

- **딕셔너리에서 키의 값을 가져오기**
  - **get(키) : 딕셔너리에서 특정 키의 값을 가져옴**

```
>>> x = {'a': 10, 'b': 20, 'c': 30, 'd': 40}
>>> x.get('a')
10
```

  - get(키, 기본값) : 딕셔너리에 키가 있을 때 해당 키의 값을 반환
    키가 없을 때는 기본값을 반환

```
>>> x.get('z', 0)
0
```

- **딕셔너리에서 키-값 쌍을 모두 가져오기**
  - 딕셔너리는 키와 값을 가져오는 다양한 메서드를 제공함
    - items: 키-값 쌍을 모두 가져옴
    - keys: 키를 모두 가져옴
    - values: 값을 모두 가져옴
  - **items() : 딕셔너리의 키-값 쌍을 모두 가져옴**

```
>>> x = {'a': 10, 'b': 20, 'c': 30, 'd': 40}
>>> x.items()
dict_items([('a', 10), ('b', 20), ('c', 30), ('d', 40)])
```

  - **keys() : 키를 모두 가져옴**

```
>>> x.keys()
dict_keys(['a', 'b', 'c', 'd'])
```

  - **values() : 값을 모두 가져옴**

```
>>> x.values()
dict_values([10, 20, 30, 40])
```

- 리스트와 튜플로 딕셔너리 만들기
  - dict.fromkeys(키리스트) : 키 리스트로 딕셔너리를 생성하며 값은 모두 None으로 저장함

```
>>> keys = ['a', 'b', 'c', 'd']
>>> x = dict.fromkeys(keys)
>>> x
{'a': None, 'b': None, 'c': None, 'd': None}
```

  - dict.fromkeys(키리스트, 값) : 키 리스트와 값을 지정하면 해당 값이 키의 값으로 저장됨

```
>>> y = dict.fromkeys(keys, 100)
>>> y
{'a': 100, 'b': 100, 'c': 100, 'd': 100}
```