Python (5)

자료형 - 딕셔너리, 집합, 부울

02-5 딕셔너리 자료형

예를 들어 사람은 "이름" = "홍길동", "생일" = "몇 월 며칠" 등과 같은 방식으로 그 사람이 가진 정보를 나타낼 수 있다.
파이썬은 이러한 대응 관계를 나타낼 수 있는 딕셔너리(dictionary) 자료형을 가지고 있다.

요즘 사용하는 대부분의 언어도 이러한 대응 관계를 나타내는 자료형을 가지고 있는데 이를 딕셔너리라고 하고, '연관 배열(associative array)'또는 '해시(hash)'라고도 한다.

○ 딕셔너리란?

딕셔너리는 Key와 Value를 한 쌍으로 하는 여러개의 데이터를 가지는 자료형이다.
예를 들어 어떤 사람에 대한 정보를 딕셔너리로 표현하면 다음과 같다.

```
person = {
     "firstName": "John",
     "lastName": "Doe",
     "age": 23,
     "eyeColor": "blue"
}
```

딕셔너리는 리스트나 튜플처럼 순차적으로(sequential) 해당 요소값을 구하지 않고 Key를 통해 Value를 얻는다.
이것이 바로 딕셔너리의 가장 큰 특징이다.

○ 딕셔너리는 어떻게 만들까?

다음은 딕셔너리의 기본 모습이다.

```
{ Key1: Value1, Key2: Value2, Key3: Value3, ... }
```

Key와 Value의 쌍 여러 개가 { }로 둘러싸여 있다. 각각의 요소는 Key: Value 형태로 이루어져 있고 쉼표(,)로 구분되어 있다.

다음 딕셔너리의 예를 살펴보자.

```
dic = {'name': 'pey', 'phone': '010-9999-1234', 'birth': '1118'}
```

위에서 Key는 각각 'name', 'phone', 'birth', 각각의 Key에 해당하는 Value는 'pey', '010-9999-1234', '1118'이 된다.

다음은 Key로 정숫값 1, Value로 문자열 'hi'를 사용한 예이다.

```
a = {1: 'hi'}
```

또한 다음 예처럼 Value에 리스트도 넣을 수 있다.

```
a = {'a': [1, 2, 3]}
```

○ 딕셔너리 쌍 추가, 삭제하기

딕셔너리 쌍을 추가 또는 삭제하는 방법을 살펴보자. 먼저 딕셔너리에 쌍을 추가해 보자.

◆ 딕셔너리 쌍 추가하기

```
a = {1: 'a'}
a[2] = 'b'
print(a)
--------------------
{1: 'a', 2: 'b'}
```

{1: 'a'} 딕셔너리에 a[2] = 'b'와 같이 입력하면 딕셔너리 a에 Key와 Value가 각각 2와 'b'인 {2: 'b'} 딕셔너리 쌍이 추가된다.

```
a['name'] = 'pey'
print(a)
--------------------
{1: 'a', 2: 'b', 'name': 'pey'}
```

딕셔너리 a에 {'name': 'pey'} 쌍이 추가되었다.

```
a[3] = [1, 2, 3]
print(a)
--------------------
{1: 'a', 2: 'b', 'name': 'pey', 3: [1, 2, 3]}
```

Key는 3, Value는 [1, 2, 3]을 가지는 한 쌍이 또 추가되었다.

◆ 딕셔너리 요소 삭제하기

```
del a[1]
print(a)
--------------------
{2: 'b', 'name': 'pey', 3: [1, 2, 3]}
```

위 예제는 딕셔너리 요소를 지우는 방법을 보여 준다.
del 함수를 사용해서 **del a[key]**를 입력하면 지정한 **Key**에 해당하는 **{Key: Value}** 쌍이 삭제된다.

또 다른 방법으로는 다음과 같이 **pop**을 사용하는 방법이다.

```
marks = { 'Physics': 67, 'Chemistry': 72, 'Math': 89 }
element = marks.pop('Chemistry')
print('Popped Marks:', element)
--------------------
Popped Marks: 72
```

딕셔너리 **pop()**의 문법은 다음과 같다

```
dictionary.pop(key[, default])
```

여기서 **default**는 옵션으로, 딕셔너리에 **key**가 없을 경우, 리턴되는 값이다.

◆ 딕셔너리를 사용하는 방법

‘딕셔너리는 주로 어떤 것을 표현하는 데 사용할까?’라는 의문이 들 것이다.
예를 들어 4명의 사람이 있다고 가정하고 각자의 특기를 표현할 수 있는 좋은 방법에 대해서 생각해 보자.
리스트나 문자열로는 표현하기가 매우 까다로울 것이다.
하지만 파이썬의 딕셔너리를 사용하면 이 상황을 표현하기가 정말 쉽다.

다음 예를 살펴보자.

```
{ "김연아": "피겨스케이팅", "류현진": "야구", "손흥민": "축구", "귀도": "파이썬" }
```

사람 이름과 특기를 한 쌍으로 묶은 딕셔너리이다.

지금껏 우리는 딕셔너리를 만드는 방법만 살펴보았는데,
딕셔너리를 제대로 활용하기 위해 알아야 할 것이 더 있다. 지금부터 하나씩 알아보자.

◆ 딕셔너리에서 Key를 사용해 Value 얻기

```
grade = {'pey': 10, 'julliet': 99}
```

```
print(grade['pey'])
print(grade['julliet'])
--------------------
10
99
```

리스트나 튜플, 문자열은 요소값을 얻고자 할 때 인덱싱이나 슬라이싱 기법 중 하나를 사용했다.
하지만 딕셔너리는 단 1가지 방법뿐이다. 그것은 바로 Key를 사용해서 Value를 구하는 방법이다.
위 예에서 'pey'라는 Key의 Value를 얻기 위해 grade['pey']를 사용한 것처럼 어떤 Key의 Value를 얻기 위해서는 '딕셔너리변수이름[Key]'를 사용해야 한다.

몇 가지 예를 더 살펴보자.

```
a = {1:'a', 2:'b'}
print(a[1])
print(a[2])
--------------------
'a'
'b'
```

먼저 a 변수에 {1: 'a', 2: 'b'} 딕셔너리를 대입하였다.
위 예에서 볼 수 있듯이 a[1]은 'a' 값을 리턴한다.

여기에서 a[1]이 의미하는 것은 리스트나 튜플의 a[1]과는 전혀 다르다.
딕셔너리 변수에서 [] 안의 숫자 1은 두 번째 요소를 나타내는 것이 아니라 Key에 해당하는 1을 나타낸다.
앞에서도 말했듯이 딕셔너리는 리스트나 튜플에 있는 인덱싱 방법을 적용할 수 없다.
따라서 a[1]은 딕셔너리 {1: 'a', 2: 'b'}에서 Key가 1인 것의 Value인 'a'를 리턴한다. a[2] 역시 마찬가지이다.

이번에는 a라는 변수에 앞의 예에서 사용한 딕셔너리의 Key와 Value를 뒤집어 놓은 딕셔너리를 대입해 보자.

```
a = {'a': 1, 'b': 2}
print(a['a'])
print(a['b'])
--------------------
1
2
```

역시 a['a'], a['b']처럼 Key를 사용해서 Value를 얻을 수 있다. 정리하면,

딕셔너리 a는 a[Key]로 Key에 해당하는 Value를 얻는다.

다음 예는 이전에 한 번 언급한 딕셔너리인데, Key를 사용해서 Value를 얻는 방법을 잘 보여 준다.

```
dic = {'name':'pey', 'phone':'010-9999-1234', 'birth': '1118'}
print(dic['name'])
```

```
print(dic['phone'])
printdic['birth']
--------------------
'pey'
'010-9999-1234'
'1118'
```

◆ 딕셔너리 만들 때 주의할 사항

딕셔너리에서 **Key**는 고유한 값이므로 중복되는 **Key** 값을 설정해 놓으면 하나를 제외한 나머지 것들이 모두 무시된다는 점에 주의해야 한다.
다음 예에서 볼 수 있듯이 동일한 **Key**가 **2**개 존재할 경우, **1: 'a'** 쌍이 무시된다.

```
a = {1:'a', 1:'b'}
print(a)
--------------------
{1: 'b'}
```

이렇게 **Key**가 중복되었을 때 **1**개를 제외한 나머지 **Key: Value** 값이 모두 무시되는 이유는 **Key**를 통해서 **Value**를 얻는 딕셔너리의 특징 때문이다.
즉, 딕셔너리에는 동일한 **Key**가 중복으로 존재할 수 없다.

또 한가지 주의해야 할 점은 Key에 리스트는 쓸 수 없다는 것이다.
하지만 튜플은 Key로 쓸 수 있다.
딕셔너리의 Key로 쓸 수 있느냐, 없느냐는 Key가 변하는(mutable) 값인지, 변하지 않는(immutable) 값인지에 달려 있다.
리스트는 그 값이 변할 수 있기 때문에 Key로 쓸 수 없다.
다음 예처럼 리스트를 Key로 설정하면 리스트를 키 값으로 사용할 수 없다는 오류가 발생한다.

```
a = {[1,2] : 'hi'}
```

단, Value에는 변하는 값이든, 변하지 않는 값이든 아무 값이나 넣을 수 있다.

○ 딕셔너리 관련 함수

딕셔너리를 자유자재로 사용하기 위해 딕셔너리가 자체적으로 가지고 있는 관련 함수를 사용해 보자.

◆ Key 리스트 만들기 - keys

```
a = {'name': 'pey', 'phone': '010-9999-1234', 'birth': '1118'}
print(a.keys())
```

```
--------------------
dict_keys(['name', 'phone', 'birth'])
```

a.keys()는 딕셔너리 a의 Key만을 모아 dict_keys 객체를 리턴한다.

dict_keys 객체는 다음과 같이 사용할 수 있다. 리스트를 사용하는 것과 별 차이는 없지만, 리스트 고유의 append, insert, pop, remove, sort 함수는 수행할 수 없다.

```
for k in a.keys():
    print(k)
--------------------
name
phone
birth
```

print(k)를 입력할 때 들여쓰기를 하지 않으면 오류가 발생하므로 주의하자.
for 문 등 반복 구문에 대해서는 뒤에서 자세히 살펴본다.

dict_keys 객체를 리스트로 변환하려면 다음과 같이 하면 된다.

```
list(a.keys())
--------------------
```

```
['name', 'phone', 'birth']
```

◆ Value 리스트 만들기 – values

```
print(a.values())
--------------------
dict_values(['pey', '010-9999-1234', '1118'])
```

Key를 얻는 것과 마찬가지 방법으로 Value만 얻고 싶다면 values 함수를 사용하면 된다. values 함수를 호출하면 dict_values 객체를 리턴한다.

◆ Key, Value 쌍 얻기 – items

```
print(a.items())
--------------------
dict_items([('name', 'pey'), ('phone', '010-9999-1234'), ('birth', '1118')])
```

items 함수는 Key와 Value의 쌍을 튜플로 묶은 값을 dict_items 객체로 리턴한다.

◆ Key: Value 쌍 모두 지우기 - clear

```
a.clear()
print(a)
--------------------
{}
```

clear 함수는 딕셔너리 안의 모든 요소를 삭제한다.

빈 리스트를 [], 빈 튜플을 ()로 표현하는 것과 마찬가지로 빈 딕셔너리도 {}로 표현한다.

◆ Key로 Value 얻기 - get

```
a = {'name': 'pey', 'phone': '010-9999-1234', 'birth': '1118'}
print(a.get('name'))
print(a.get('phone'))
--------------------
'pey'
'010-9999-1234'
```

get(x) 함수는 x라는 Key에 대응되는 Value를 리턴한다. 앞에서 살펴보았듯이 a.get('name')은 a['name']을 사용했을 때와 동일한 결괏값을 리턴한다.

다만, 다음 예제에서 볼 수 있듯이 a['nokey']처럼 딕셔너리에 존재하지 않는 키로 값을 가져오려고 할 경우,
a['nokey'] 방식은 오류를 발생시키고
a.get('nokey') 방식은 None을 리턴한다는 차이가 있다.

여기에서 None은 '거짓'이라는 뜻이라고만 알아 두자.

딕셔너리 안에 찾으려는 Key가 없을 경우, 미리 정해 둔 디폴트 값을 대신 가져오게 하고 싶을 때는 get(x, '디폴트 값')을 사용하면 편리하다.

```
print(a.get('nokey', 'foo'))
--------------------
'foo'
```

딕셔너리 a에는 'nokey'에 해당하는 Key가 없다. 따라서 디폴트 값인 'foo'를 리턴한다.

◆ 해당 Key가 딕셔너리 안에 있는지 조사하기 - in

```
a = {'name':'pey', 'phone':'010-9999-1234', 'birth': '1118'}
```

```
print('name' in a)
print('email' in a)
--------------------
True
False
```

'name' 문자열은 **a** 딕셔너리의 **Key** 중 하나이다.
따라서 **'name' in a**를 호출하면 참**(True)**을 리턴한다.
이와 반대로 **'email'**은 **a** 딕셔너리 안에 존재하지 않는 **Key**이므로 거짓**(False)**을 리턴한다.

02-6 집합 자료형

집합(set)은 집합에 관련된 것을 쉽게 처리하기 위해 만든 자료형이다.

○ 집합 자료형은 어떻게 만들까?

집합 자료형은 다음과 같이 set 키워드를 사용해 만들 수 있다.

```
s1 = set([1, 2, 3])
print(s1)
--------------------
{1, 2, 3}
```

위와 같이 set()의 괄호 안에 리스트를 입력하여 만들거나 다음과 같이 문자열을 입력하여 만들 수도 있다.

```
s2 = set("Hello")
print(s2)
```

```
--------------------
{'e', 'H', 'l', 'o'}
```

비어 있는 집합 자료형은 **s = set()**로 만들 수 있다.

○ 집합 자료형의 특징

그런데 위에서 살펴본 **set("Hello")**의 결과가 좀 이상하지 않은가?
분명 **"Hello"** 문자열로 **set** 자료형을 만들었는데 생성된 자료형에는 **l** 문자가 하나 빠져 있고 순서도 뒤죽박죽이다.
그 이유는 **set**에 다음과 같은 **2**가지 특징이 있기 때문이다.

- 중복을 허용하지 않는다.
- 순서가 없다(**Unordered**).

set은 중복을 허용하지 않는 특징 때문에 데이터의 중복을 제거하기 위한 필터로 종종 사용된다.

리스트나 튜플은 순서가 있기(**ordered**) 때문에 인덱싱을 통해 요솟값을 얻을 수 있지만, **set** 자료형은 순서가 없기(**unordered**) 때문에 인덱싱을 통해 요솟값을 얻을 수 없다.
이는 마치 **02-5**에서 살펴본 딕셔너리와 비슷하다. 딕셔너리 역시 순서가 없는 자료형이므로 인덱싱을 지원하지 않는다.

만약 set 자료형에 저장된 값을 인덱싱으로 접근하려면 다음과 같이 리스트나 튜플로 변환한 후에 해야 한다.

```
s1 = set([1, 2, 3])

l1 = list(s1)
print(l1)
print(li[0])

ti = tuple(s1)
print(t1)
print(t1[0])
--------------------
[1, 2, 3]
1
(1, 2, 3)
1
```

○ 교집합, 합집합, 차집합 구하기

set 자료형을 정말 유용하게 사용하는 경우는 교집합, 합집합, 차집합을 구할 때이다.

먼저 다음과 같이 2개의 set 자료형을 만든 후 따라 해 보자.
s1에는 1부터 6까지의 값, s2에는 4부터 9까지의 값을 주었다.

```
s1 = set([1, 2, 3, 4, 5, 6])
s2 = set([4, 5, 6, 7, 8, 9])
```

◆ 교집합 구하기

s1과 s2의 교집합을 구해 보자. 다음과 같이 '&'를 이용하면 교집합을 간단히 구할 수 있다.

```
print(s1 & s2)
--------------------
{4, 5, 6}
```

또는 다음과 같이 intersection 함수를 사용해도 결과는 동일하다.

```
print(s1.intersection(s2))
--------------------
{4, 5, 6}
```

s2.intersection(s1)을 사용해도 결과는 동일하다.

◆ 합집합 구하기

이번에는 합집합을 구해 보자. '|'를 사용하면 합집합을 구할 수 있다.
이때 **4, 5, 6**처럼 중복해서 포함된 값은 **1**개씩만 표현된다.

```
print(s1 | s2)
--------------------
{1, 2, 3, 4, 5, 6, 7, 8, 9}
```

union 함수를 사용해도 된다.

```
print(s1.union(s2))
--------------------
{1, 2, 3, 4, 5, 6, 7, 8, 9}
```

s2.union(s1)을 사용해도 결과는 동일하다.

◆ 차집합 구하기

마지막으로 차집합을 구해 보자. -(빼기)를 사용하면 차집합을 구할 수 있다.

```
print(s1 – s2)
print(s2 - s1)
--------------------
{1, 2, 3}
{8, 9, 7}
```

difference 함수를 사용해도 차집합을 구할 수 있다.

```
print(s1.difference(s2))
print(s2.difference(s1))
--------------------
{1, 2, 3}
{8, 9, 7}
```

○ 집합 자료형 관련 함수

◆ 값 1개 추가하기 – add

이미 만들어진 set 자료형에 값을 추가할 수 있다. 1개의 값만 추가 할 때는 다음과 같이 하면 된다.

```
s1 = set([1, 2, 3])
s1.add(4)
print(s1)
--------------------
{1, 2, 3, 4}
```

◆ 값 여러 개 추가하기 – update

여러 개의 값을 한꺼번에 추가(update)할 때는 다음과 같이 하면 된다.

```
s1 = set([1, 2, 3])
s1.update([4, 5, 6])
print(s1)
```

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

◆ 특정 값 제거하기 – remove

특정 값을 제거하고 싶을 때는 다음과 같이 하면 된다.

```
s1 = set([1, 2, 3])
s1.remove(2)
print(s1)
--------------------
{1, 3}
```

02-7 불 자료형

불(bool) 자료형이란 참(True)과 거짓(False)을 나타내는 자료형이다. 불 자료형은 다음 2가지 값만을 가질 수 있다.

- True: 참을 의미한다.
- False: 거짓을 의미한다.

True나 False는 파이썬의 예약어로, 첫 문자를 항상 대문자로 작성해야 한다.

❍ 불 자료형은 어떻게 사용할까?

다음과 같이 변수 a에는 True, 변수 b에는 False를 지정해 보자.

```
a = True
b = False
```

따옴표로 감싸지 않은 문자열을 변수에 지정해서 오류가 발생할 것 같지만, 잘 실행된다. type 함수를 변수 a와 b에 사용하면 두 변수의 자료형이 bool로 지정된 것을 확인할 수 있다

```
print(type(a))
print(type(b))
--------------------
<class 'bool'>
<class 'bool'>
```

type(x)는 **x**의 자료형을 확인하는 파이썬의 내장 함수이다.

불 자료형은 조건문의 리턴값으로도 사용된다. 조건문에 대해서는 **if** 문에서 자세히 배우겠지만 잠시 살펴보고 넘어가자.

```
print(1 == 1)
--------------------
True
```

1 == 1 은 **‘1**과 **1**이 같은가**?’**를 묻는 조건문이다. 이런 조건문은 결과로 **True** 또는 **False**에 해당하는 불 자료형을 리턴한다. **1**과 **1**은 같으므로 **True**를 리턴한다.

```
print(2 > 1)
--------------------
True
```

2는 1보다 크므로 2 > 1 조건문은 참이다. 즉, True를 리턴한다.

```
print(2 < 1)
--------------------
False
```

2는 1보다 작지 않으므로 2 < 1 조건문은 거짓이다. 즉, False를 리턴한다.

○ 자료형의 참과 거짓

'자료형에 참과 거짓이 있다?'라는 말이 조금 이상하게 들리겠지만, 참과 거짓은 분명히 있다. 이는 매우 중요한 특징이며 실제로도 자주 쓰인다.

자료형의 참과 거짓을 구분하는 기준은 다음과 같다.

값	참 or 거짓
"python"	참
""	거짓

[1, 2, 3]	참
[]	거짓
(1, 2, 3)	참
()	거짓
{'a': 1}	참
{}	거짓
1	참
0	거짓
None	거짓

문자열, 리스트, 튜플, 딕셔너리 등의 값이 비어 있으면("", [], (), {}) 거짓이 되고, 비어 있지 않으면 참이 된다.
숫자에서는 그 값이 0일 때 거짓이 된다.
위 표를 보면 None이 있는데, 이것에 대해서는 뒤에서 학습할 것이다.
그저 None은 거짓을 뜻한다는 것만 알아 두자.

다음 예를 보고 자료형의 참과 거짓이 프로그램에서 어떻게 쓰이는지 간단히 알아보자.

```
a = [1, 2, 3, 4]
while a:
    print(a.pop())
--------------------
```

```
4
3
2
1
```

먼저 a = [1, 2, 3, 4]라는 리스트를 만들었다.
while 문은 03장에서 자세히 다루지만, 간단히 알아보면 다음과 같다.
조건문이 참인 동안 조건문 안에 있는 문장을 반복해서 수행한다.

```
while 조건문:
    수행할_문장
```

즉, 위 예를 보면 a가 참인 경우, a.pop()를 계속 실행하여 출력하라는 의미이다.
a.pop() 함수는 리스트 a의 마지막 요소를 끄집어 내는 함수이므로 리스트 안에 요소가 존재하는 한(a가 참인 동안) 마지막 요소를 계속 끄집어 낼 것이다.
결국 더 이상 끄집어 낼 것이 없으면 a가 빈 리스트([])가 되어 거짓이 된다.
따라서 while 문에서 조건문이 거짓이 되므로 while 문을 빠져나가게 된다.
이는 파이썬 프로그래밍에서 매우 자주 사용하는 기법 중 하나이다.

위 예가 너무 복잡하다고 생각하면, 다음 예를 보면 쉽게 이해될 것이다.

```
if []:
    print("참")
else:
    print("거짓")
--------------------
거짓
```

[]는 앞의 표에서 볼 수 있듯이 비어 있는 리스트이므로 거짓이다. 따라서 "거짓"이라는 문자열이 출력된다.
if 문에 대해서 잘 모르는 독자라도 위 문장을 해석하는 데는 무리가 없을 것이다.

    if 문에 대해서는 뒤에서 자세히 다룬다.

다른 예도 하나만 더 살펴보자.

```
if [1, 2, 3]:
    print("참")
else:
    print("거짓")
--------------------
참
```

이 소스 코드를 해석해 보면 다음과 같다

```
만약 [1, 2, 3]이 참이면 "참"이라는 문자열을 출력하고, 그렇지 않으면 "거짓"이라는 문자열을 출력하라.
```

[1, 2, 3]은 요솟값이 있는 리스트이므로 참이다. 따라서 "참"을 출력한다.

○ 불 연산

자료형에 참과 거짓이 있다는 것을 이제 알게 되었다. bool 함수를 사용하면 자료형의 참과 거짓을 보다 정확하게 식별할 수 있다.

다음 예제를 따라 해 보자.

```
print(bool('python'))
--------------------
True
```

'python' 문자열은 비어 있지 않으므로 bool 연산의 결과로 불 자료형인 True를 리턴한다.

```
print(bool(''))
--------------------
False
```

'' 문자열은 비어 있으므로 **bool** 연산의 결과로 불 자료형인 **False**를 리턴한다.

앞에서 알아본 몇 가지 예제를 더 수행해 보자.

```
bool([1, 2, 3])
bool([])
bool(0)
bool(3)
-------------------
True
False
False
True
```

앞에서 알아본 것과 동일한 참과 거짓에 대한 결과를 리턴하는 것을 확인할 수 있다.

02-8 변수

변수(Variables)란 값을 저장하는 공간이라 생각하면 된다.

○ 변수는 어떻게 만들까?

우리는 앞에서 이미 변수를 사용해 왔다. 다음 예와 같은 a, b, c를 '변수'라고 한다.

```
a = 1
b = "python"
c = [1, 2, 3]
```

변수를 만들 때는 위 예처럼 '=' 할당연산자(assignment)를 사용한다.

```
변수이름 = 변수값
```

다른 프로그래밍 언어인 C나 JAVA에서는 변수를 만들 때 자료형의 타입을 직접 지정해야 한다. 하지만 파이썬은 변수에 저장된 값을

스스로 판단하여 자료형의 타입을 지정하기 때문에 더 편리하다.

○ 변수란?

파이썬에서 사용하는 변수는 객체를 가리키는 것이라고도 말할 수 있다.
객체란 우리가 지금까지 보아 온 자료형의 데이터(값)와 같은 것을 의미하는 말이다(객체에 대해서는 뒤에서 자세하게 공부한다).

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

만약 위 코드처럼 a = [1, 2, 3]이라고 하면 [1, 2, 3] 값을 가지는 리스트 데이터(객체)가 자동으로 메모리에 생성되고 변수 a는 [1, 2, 3] 리스트가 저장된 메모리의 주소를 가리키게 된다.

메모리란 컴퓨터가 프로그램에서 사용하는 데이터를 기억하는 공간을 말한다.

a 변수가 가리키는 메모리의 주소는 다음과 같이 확인할 수 있다.

```
a = [1, 2, 3]
print(id(a))
--------------------
```

```
4303029896
```

id는 변수가 가리키고 있는 객체의 주소 값을 리턴하는 파이썬의 내장 함수이다.
즉, 여기에서 필자가 만든 변수 a가 가리키는 [1, 2, 3] 리스트의 주소 값은 4303029896이라는 것을 알 수 있다.

○ 리스트를 복사하고자 할 때

이번에는 리스트 자료형에서 가장 혼동하기 쉬운 '복사'에 대해 설명한다. 다음 예를 통해 알아보자.

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

b 변수에 a 변수를 대입하면 어떻게 될까?
b와 a는 같은 걸까, 다른 걸까?
결론부터 말하면 b는 a와 완전히 동일하다고 할 수 있다.
다만 [1, 2, 3]이라는 리스트 객체를 참조하는 변수가 a 변수 1개에서 b 변수가 추가되어 2개로 늘어났다는 차이만 있을 뿐이다.

id 함수를 사용하면 이러한 사실을 확인할 수 있다.

```
print(id(a))
print(id(b))
--------------------
4303029896
4303029896
```

id(a)의 값이 **id(b)**의 값과 동일하다는 것을 확인할 수 있다.
즉, **a**가 가리키는 대상과 **b**가 가리키는 대상이 동일하다는 것을 알 수 있다.
동일한 객체를 가리키고 있는지에 대해서 판단하는 파이썬 명령어 **is**를 다음과 같이 실행해도 역시 참을 리턴해 준다.

```
print(a is b)           # a와 b가 가리키는 객체가 같을까?
--------------------
True
```

이제 다음 예를 계속 수행해 보자.

```
a[1] = 4
print(a)
print(b)
--------------------
[1, 4, 3]
```

```
[1, 4, 3]
```

a 리스트의 두 번째 요소를 값 4로 바꾸었더니 a만 바뀌는 것이 아니라 b도 똑같이 바뀌었다.
그 이유는 앞에서 살펴본 것처럼 a, b 모두 동일한 리스트를 가리키고 있기 때문이다.

그렇다면 b 변수를 생성할 때 a 변수의 값을 가져오면서 a와는 다른 주소를 가리키도록 만들 수는 없을까?
다음 2가지 방법이 있다.

1. [:] 이용하기

첫 번째 방법은 다음과 같이 리스트 전체를 가리키는 [:]을 사용해서 복사하는 것이다.

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

print(a)
print(b)
--------------------
[1, 4, 3]
[1, 2, 3]
```

위 예에서 볼 수 있듯이 **a** 리스트 값을 바꾸더라도 **b** 리스트에는 아무런 영향이 없다.

2. copy 모듈 이용하기

두 번째 방법은 **copy** 모듈을 사용하는 것이다.
다음 예를 보면 **from copy import copy**라는 처음 보는 형태의 문장이 나오는데, 이것은 뒤에 설명할 파이썬 모듈 부분에서 자세히 다룬다.
여기에서는 단순히 **copy** 함수를 쓰기 위해서 사용하는 것이라고만 알아 두자.

```
from copy import copy
a = [1, 2, 3]
b = copy(a)
```

위 예에서 **b = copy(a)**는 **b = a[:]**과 동일하다.

두 변수의 값은 같지만, 서로 다른 객체를 가리키고 있는지 다음과 같이 확인해 보자.

```
print(b is a)
--------------------
False
```

위 예에서 b is a가 False를 리턴하므로 b와 a가 가리키는 객체는 서로 다르다는 것을 알 수 있다.

◆ copy 함수 사용하기

다음처럼 리스트 자료형의 자체 함수인 copy 함수를 사용해도 copy 모듈을 사용하는 것과 동일한 결과를 얻을 수 있다.

```
a = [1, 2, 3]
b = a.copy()
print(b is a)
--------------------
False
```

◆ 변수를 만드는 여러 가지 방법

다음과 같이 튜플로 a, b에 값을 대입할 수 있다.

```
a, b = ('python', 'life')
```

이 방법은 다음 예문과 완전히 동일하다.

```
(a, b) = 'python', 'life'

혹은

a, b = 'python', 'life'
```

튜플 부분에서도 언급했지만, 튜플은 괄호를 생략해도 된다.

다음처럼 리스트로 변수를 만들 수도 있다.

```
[a, b] = ['python', 'life']

혹은

a, b = ['python', 'life']
```

또한 여러 개의 변수에 같은 값을 대입할 수도 있다.

```
a = b = 'python'
```

파이썬에서는 위 방법을 사용하여 두 변수의 값을 매우 간단하게 바꿀 수 있다.

```
a = 3
b = 5
a, b = b, a
print(a)
print(b)
--------------------
5
3
```

처음에 a에 값 3, b에는 값 5가 대입되어 있었지만 a, b = b, a 문장을 수행한 후에는 그 값이 서로 바뀌었다는 것을 확인할 수 있다.