6장 리스트, 튜플, 딕셔너리

데이터 사이언스에서는 주로 대용량의 데이터를 다룬다. 파이썬 프로그램에서 이러한 대용량의 데이터를 다루기 위해서는 많은 수의 데이터를 한꺼번에 저장하는 컨테이너와 같은 것이 있으면 매우 유용하다. 이번 장에서 학습할 리스트(list), 튜플(tuple), 딕셔너리(dictionary)는 많은 데이터를 한꺼번에 저장하고 찾아주는 매우 편리한 자료구조이며, 파이썬에서 매우 활발히 사용된다.

6.1 리스트(list)

리스트를 이용하면 여러 개의 값을 하나의 리스트 안에 저장이 가능하다. 리스트를 이용하지 않고 기존의 변수를 이용하려면 저장하고자 하는 값의 개수만큼 변수를 선언해 주어야 하고 각각의 이름도 따로 지정해야 하는 불편함이 있다. 리스트를 생성하는 방법은 다음과 같다. 리스트는 _____ 안에 값을 선언한다.

```
리스트명 = [값1, 값2, 값3, ...]
```

다음 코드는 값 5개를 담은 정수형 리스트를 생성한 것이다.

```
aa = [10, 20, 30, 40, 50]
```

리스트를 사용할 경우 각각의 값에 대한 이름은 첨자(index)를 이용한다. 리스트의 첫 번째 값에 대한 변수 이름은 aa[0]이 되고, 두 번째는 aa[1]이 된다. 위의 예제에서처럼 리스트 aa를 선언하면 변수 aa[0], aa[1], aa[2], aa[3], aa[4]가 한꺼번에 선언된 것과 같다고 볼 수 있다. 여기서 주의할 점은 첨자는 1이 아닌 0부터 시작한다는 점이다.

리스트에 값을 추가할 때에는 _____를 이용한다. append() 함수는 리스트에 사용할 수 있는 멤버 함수인데, 괄호 안에 값을 넣으면 해당 리스트에 입력한 값이 추가된다. append() 함수는 리스트의 멤버 함수(member function) 또

는 메소드(method)이기 때문에 ‘리스트이름.append(값)’ 의 형식으로 사용한다. 아래 예제는 비어있는 리스트를 생성하고 3개의 값을 추가한 후 리스트를 출력하는 프로그램이다.

```
aa=[]
aa.append(1)
aa.append(2)
aa.append(3)
print(aa)
```

```
<실행 결과>
[1, 2, 3]
```

리스트는 정수만으로 구성할 수도 있고, 실수만으로 구성할 수도 있다. 물론 문자열만으로도 구성할 수 있다. 그리고 다른 언어의 배열과 다른 점은 아래와 같이 _____ 수도 있다.

```
aa=[10, 12.3, "파이썬", True]
print(aa)
```

```
<실행 결과>
[10, 12.3, '파이썬', True]
```

● 리스트 값으로의 접근 방법

생성된 리스트는 ____로 접근한다. 여기서 첨자는 0부터 시작하는 정수 값뿐만 아니라 다양한 방법으로 접근 가능하다. 다음은 리스트 값에 접근하는 방법에 대한 예제이다.

```
aa=[10, 20, 30, 40]
print(aa[-1])
print(aa[0:3])
print(aa[2:4])
print(aa[2:])
```

```
<실행 결과>
40
[10, 20, 30]
[30, 40]
[30, 40]
```

첫 번째, 음수 값을 첨자로 쓸 수 있다. aa[-1]은 제일 뒤에서 _____란 의미이다. 그래서 리스트에서 제일 마지막 값인 40이 출력 되었다. 그 다음 a[0:3]은 0번 첨자부터 2번 첨자까지를 의미한다. 출력 결과에 대괄호가 있음을 확인할 수 있는데 a[0:3]는 0번부터 2번까지의 _____임을 알 수 있다. 참고로 aa[0]은 리스트가 아닌 값이다. 첨자 0:3에서 마지막 3은 첨자로 포함되지 않는다는 점도 주의하자. 그래서 aa[2:4]는 aa[2], aa[3]으로 이루어진 _____이다. 예제의 마지막 줄에서처럼 숫자를 생략하기도 하는데 마지막까지 또는 처음부터를 의미한다.

리스트끼리 ____과 ____ 연산이 가능하다. 두 리스트의 덧셈 연산은 리스트 두 개를 이어 붙인 새로운 리스트를 생성함을 의미하고, 리스트에 정수를 곱한 연산은 해당 리스트를 숫자만큼 반복하여 이어 붙인 리스트를 생성함을 의미한다. 다음은 리스트 연산에 대한 예제이다.

```
aa=[10, 20, 30, 40]
bb=[100, 200, 300]
cc=aa+bb
dd=aa*3
print(cc)
print(dd)
```

```
<실행 결과>
[10, 20, 30, 40, 100, 200, 300]
[10, 20, 30, 40, 10, 20, 30, 40, 10, 20, 30, 40]
```

● 리스트 값의 변경

리스트 값을 변경하는 방법은 여러 가지가 있는데, 다음은 리스트의 값을 다른 값으로 변경하는 예제이다.

```
aa=[10, 20, 30, 40]
aa[1] = -100
print(aa)
aa[2:3] = [300, 400]
print(aa)
```

```
<실행 결과>
[10, -100, 30, 40]
```

```
[10, -100, 300, 400, 40]
```

위의 예제에서 보듯이 두 번째 값을 변경하기 위해서는 해당 변수 이름인 aa[1]을 통하여 간단히 변경하면 된다. 만약 3번째 자리에 있는 aa[2] 값 대신 새로 두 개의 값을 추가하고자 한다면 위의 예제의 aa[2:3]처럼 범위를 나타내는 첨자를 이용하여 리스트를 대입하면 된다. 그러면 3번째 자리의 30 대신 300과 400이 추가되었음을 확인할 수 있다.

리스트내의 값을 ____하는 방법을 다음의 예제로 설명하겠다.

```
aa=[10, 20, 30, 40]
del(aa[1])
print(aa)

bb=[10, 20, 30, 40]
bb[2:4] = []
print(bb)

bb=[]
print(bb)
```

<실행 결과>

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

del() 함수는 괄호 안 변수의 값을 리스트에서 삭제한다. 그래서 aa[1]에 해당하는 20이 리스트에서 삭제되었다. 리스트 안에서 특정 범위의 값들을 삭제하기 위해서는 위의 예에서처럼 해당 범위에 빈 리스트를 대입하면 된다. 또한 del(bb[2:4])처럼 del() 함수를 이용해도 동일한 결과가 나온다.

● 리스트 조작 함수

del(), append()와 같은 리스트 조작 함수들을 아래와 같이 표로 정리하였다.

함수	설명	사용법

append()	리스트 맨 뒤에 항목을 추가한다.	리스트명.append(값)
pop()	리스트 맨 뒤의 항목을 빼낸다. 해당 항목은 삭제된다.	리스트명.pop()
sort()	리스트의 항목을 정렬한다.	리스트명.sort()
index()	지정한 값을 찾아 해당 위치를 반환한다.	리스트명.index(찾을값)
insert()	지정된 위치에 값을 삽입한다.	리스트명.insert(위치,값)
remove()	리스트에서 지정한 값을 삭제한다. 지정한 값이 리스트 안에서 여러 개면 천 번째 값만 지운다.	리스트명.remove(지울값)
count()	리스트에서 해당 값의 개수를 세어서 반환한다.	리스트명.count(찾을값)
del()	리스트에서 해당 위치의 항목을 삭제한다.	del(리스트명[위치])
len()	리스트에 포함된 전체 항목의 개수를 반환한다.	len(리스트명)

6.2 튜플(tuple)

튜플은 리스트와 매우 비슷하다. 약간 다른 점은 튜플은 _____로 생성한다. 또한 튜플은 값을 수정할 수 없으며, ____만 가능하다. 따라서 읽기 전용 값을 저장할 때 주로 사용한다.

튜플을 만드는 방법은 다음과 같다.

```
튜플명 = (값1, 값2, 값3, ...)
튜플명 = 값1, 값2, 값3, ...
```

튜플은 소괄호를 생략해도 된다. 튜플은 읽기 전용이기 때문에 다음 코드는 첫 번째 선언 부분을 제외하고 모두 ____가 발생한다.

```
aa=(10, 20, 30)
aa.append(40)
aa[0] = 100
del(aa[0])
```

튜플 항목에 접근하는 방법은 리스트와 동일하다. '튜플명[위치]'를 사용한다. 위치에 대한 범위를 나타내기 위하여 콜론(:)을 이용하는 방식도 동일하다. 또한 튜플 끼리의 덧셈 및 정수 곱셈 연산도 가능하다.

6.3 딕셔너리(dictionary)

딕셔너리는 말 그대로 사전 구조이다. 영어 사전에서 한 항목은 단어와 단어의 뜻이 쌍을 이루어 존재하는 것과 비슷하게 파이썬에서 딕셔너리는 한 항목이 키와 값의 쌍이 각 항목으로 존재하는 구조이다. 만약 여러분이 C++의 STL(standard template library)을 사용한 경험이 있다면, STL의 map과 유사한 구조이다.

딕셔너리는 _____로 묶어 구성하며, 키(key)와 값(value)의 쌍으로 다음과 같이 구성된다.

```
딕셔너리명 = {키1:값1, 키2:값2, 키3:값3, ...}
```

다음과 같이 딕셔너리를 만들 수 있다. 키를 ‘apple’, ‘orange’, ‘mango’로 하고 각각의 값을 ‘사과’, ‘오렌지’, ‘망고’로 만들었다.

```
fruits={'apple':'사과', 'orange':'오렌지','mango':'망고'}
print(fruits)
```

```
<실행 결과>
{'apple': '사과', 'orange': '오렌지', 'mango': '망고'}
```

딕셔너리에서는 ‘딕셔너리명[키]=값’ 형식으로 항목을 추가할 수 있다. 위에서 생성한 fruits에 과일을 다음과 같이 추가할 수 있다.

```
fruits={'apple':'사과', 'orange':'오렌지','mango':'망고'}
fruits['cherry']='체리'
print(fruits)
```

```
<실행 결과>
{'apple': '사과', 'orange': '오렌지', 'mango': '망고', 'cherry': '체리'}
```

또한 딕셔너리는 여러 정보를 하나의 변수로 표현할 때 유용하다. 예를 들어 학생 한명에는 학번, 이름, 학년과 같은 정보가 있을 것인데, 이를 딕셔너리로 표현하면 다음과 같다.

```
student1={'학번':20191101, '이름':'박원주','학년':2}
print(student1)
```
<실행 결과>
```
{'학번': 20191101, '이름': '박원주', '학년': 2}
```

위의 딕셔너리에 연락처를 추가는 방법은 다음과 같다.

```
student1={'학번':20191101, '이름':'박원주','학년':2}
student1['연락처']='010-1234-5678'
print(student1)
```
<실행 결과>
```
{'학번': 20191101, '이름': '박원주', '학년': 2, '연락처': '010-1234-5678'}
```

값을 수정하는 방법 또한 새로이 입력하는 방식과 동일하다. 기존에 동일한 키가 이미 등록되어 있으면 값이 수정된다.

```
student1={'학번':20191101, '이름':'박원주','학년':2}
student1['학년']=3
print(student1)
```
<실행 결과>
```
{'학번': 20191101, '이름': '박원주', '학년': 3}
```

딕셔너리 항목의 삭제는 ‘del(딕셔너리명[키])’ 형식으로 할 수 있다. 다음 코드는 student1의 학년을 삭제한다.

```
student1={'학번':20191101, '이름':'박원주','학년':2}
del(student1['학년'])
print(student1)
```
<실행 결과>
```
{'학번': 20191101, '이름': '박원주'}
```

● 딕셔너리에서 값에로의 접근 방법

‘딕셔너리명.get(키)’ 함수를 사용하여 키를 통해 값을 가져올 수 있다.

```
student1={'학번':20191101, '이름':'박원주','학년':2}
print(student1.get('이름'))
```

```
<실행 결과>
박원주
```

딕셔너리와 관련된 함수 중 ‘딕셔너리명.keys()’는 딕셔너리의 모든 키를 ____ __ 형태로 반환한다.

```
student1={'학번':20191101, '이름':'박원주','학년':2}
print(student1.keys())
```
```
<실행 결과>
dict_keys(['학번', '이름', '학년'])
```

위의 결과는 리스트에 해당하나 dict_keys()를 없애고 순수한 리스트로 변환하기 위해서는 list() 함수를 사용하면 된다.

```
student1={'학번':20191101, '이름':'박원주','학년':2}
print(list(student1.keys()))
```
```
<실행 결과>
['학번', '이름', '학년']
```

위와 유사하게 ‘딕셔너리명.values()’는 딕셔너리의 모든 값을 리스트 형태로 반환한다.

```
student1={'학번':20191101, '이름':'박원주','학년':2}
print(student1.values())
```
```
<실행 결과>
dict_values([20191101, '박원주', 2])
```

‘딕셔너리명.items()’함수를 사용하면 튜플 형태를 얻을 수 있다.

```
student1={'학번':20191101, '이름':'박원주','학년':2}
print(student1.items())
```
```
<실행 결과>
dict_items([('학번', 20191101), ('이름', '박원주'), ('학년', 2)])
```

딕셔너리 안에 해당 키가 있는지 없는지는 ‘키 in 딕셔너리명’을 통해 확인할 수 있다.

```
student1={'학번':20191101, '이름':'박원주','학년':2}
print('이름' in student1)
print('연락처' in student1)
```

```
<실행 결과>
True
False
```

다음은 for 문을 활용해 딕셔너리의 모든 값을 출력하는 예제이다. 여기서 '딕셔너리명.keys()'는 _____이며, for 문의 'in 리스트명'은 리스트 안의 항목 개수만큼 반복됨을 상기하자.

```
student1={'학번':20191101, '이름':'박원주','학년':2}
student1['연락처'] = '010-123-4567'

for key in student1.keys():
    print("%s ==> %s" % (key, student1[key]))
```

```
<실행 결과>
학번 ==> 20191101
이름 ==> 박원주
학년 ==> 2
연락처 ==> 010-123-4567
```

6.4 리스트의 활용

● 동시에 여러 리스트에 접근하는 방법

zip() 함수를 사용하면 동시에 여러 리스트에 접근할 수 있다.

```
list_a=['a1', 'a2', 'a3', 'a4']
list_b=['b1', 'b2', 'b3']
for a, b in zip(list_a, list_b):
    print(a, '==>', b)
```

```
<실행 결과>
a1 ==> b1
a2 ==> b2
a3 ==> b3
```

예제에서 보듯이 zip() 함수는 두 리스트 중 개수가 작은 것까지만 묶어준다. zip() 함수를 이용하여 아래와 같이 두 리스트를 튜플이나 딕셔너리로 짝지어 생성할 수도 있다.

```
list_a=['a1', 'a2', 'a3', 'a4']
list_b=['b1', 'b2', 'b3']
tupList = list(zip(list_a, list_b))
dic = dict(zip(list_a, list_b))
print(tupList)
print(dic)
```

```
<실행 결과>
[('a1', 'b1'), ('a2', 'b2'), ('a3', 'b3')]
{'a1': 'b1', 'a2': 'b2', 'a3': 'b3'}
```

● 리스트 복사

리스트의 복사할 때에는 주의하여야 한다. 동일한 메모리 공간을 공유하는 ____ 복사가 있고, 실제 종이를 복사하는 것처럼 똑같은 리스트를 하나 더 만드는 ____ 복사가 있다.

```
oldList = ['아메리카노','까페라떼','카라멜마끼아또']
newList = oldList
print(newList)
oldList[2]='에스프레소'
oldList.append('다방커피')
print(newList)
```

```
<실행 결과>
['아메리카노', '까페라떼', '카라멜마끼아또']
['아메리카노', '까페라떼', '에스프레소', '다방커피']
```

위의 예제를 살펴보면 두 번째 줄에서 newList는 oldList를 복사하였고, 그 이후는 oldList의 한 항목을 변경하고 새로운 항목을 하나 추가하였다. newList를 통해서는 아무것도 하지 않은 것을 확인할 수 있다. 그러나 마지막 줄에서 newList의 내용을 출력한 결과를 보면 리스트 안의 내용이 변경이 된 것을 확인할 수 있다. 이 말은 newList와 oldList가 이름만 다른 동일한 리스트이며 동일한 메모리 공간을 공유한다는 것을 알 수 있다. 이러한 것을 얕은 복사라고 한다. 다르게는 newList는 oldList를 참조한다고 표현하기도 한다. 만약 얕은 복사를 원

하는 것이 아니라면 다음과 같이 작성하여야 한다.

```
oldList = ['아메리카노','까페라떼','카라멜마끼아또']
newList = oldList[:]
print(newList)
oldList[2]='에스프레소'
oldList.append('다방커피')
print(newList)
```

```
<실행 결과>
['아메리카노', '까페라떼', '카라멜마끼아또']
['아메리카노', '까페라떼', '카라멜마끼아또']
```

위와 같이 작성하면 newList는 oldList와는 다른 공간에 내용일 동일한 새로운 리스트를 복사한다. 이러한 복사를 깊은 복사라 한다. 위의 코드를 보면 oldList를 변경하여도 newList는 영향을 받지 않는 것을 확인할 수 있다.

● 문자열과 리스트

문자열은 리스트와 비슷한 부분이 많다. 문자열은 따옴표로 묶어서 사용하지만 리스트에 접근하는 것과 동일한 방법으로도 접근이 가능하다.

```
ss = "안녕하세요!"
print(ss[0])
print(ss[1:3])
print(ss[2:])
```

```
<실행 결과>
안
녕하
하세요!
```

리스트에서 사용하는 더하기 연산 및 정수 곱하기 연산도 동일하게 사용할 수 있다.

```
ss = "안녕" + "하세요!"
print(ss)
ss = "안녕" * 3
print(ss)
```

```
<실행 결과>
안녕하세요!
안녕안녕안녕
```

또한 리스트에서 사용하는 len() 함수도 문자열의 길이를 셀 때 자주 사용한다.

```
ss = "안녕" + "하세요!"
print(len(ss))
```

```
<실행 결과>
6
```

문자열은 리스트처럼 사용되기 때문에 for 문에서 in 다음에 다음과 같이 활용할 수 있다.

```
ss = "안녕하세요!"
for char in ss:
    print(char)
```

```
<실행 결과>
안
녕
하
세
요
!
```