

따라하며 배우는

파이썬과 데이터 과학



7장 데이터를 리스트와 튜플로 묶어보자

이 장에서 배울 것들

- 여러 개의 데이터를 하나로 묶어서 처리하는 리스트와 튜플을 배워 보아요.
- 리스트나 튜플의 항목들을 인덱스를 이용해서 접근해 보아요.
- 리스트는 가변속성을 갖고 튜플은 불변속성을 가져요.
- 리스트나 튜플 같이 데이터가 연속해서 나타나는 것을 시퀀스라고 해요.
- 시퀀스의 일부를 잘라내는 슬라이싱을 익혀 보아요.
- 슬라이싱을 이용하여 효율적으로 데이터를 다루도록 연습해 보아요.
- 복잡한 기능의 자료형을 설계하고 만드는 객체지향 개념을 살펴 보아요.

7.1 리스트는 무엇이고 왜 필요한가

- 여러 개의 데이터를 하나로 묶어서 저장하는 것이 필요하다. 파이썬에서는 **리스트**list와 **딕셔너리**dictionary를 이용하여 여러 개의 데이터를 한꺼번에 저장하고 처리할 수 있다.
- 이제까지 우리는 변수를 사용하여 데이터를 저장하고 처리하였다. 예를 들어서 어떤 사람의 키는 다음과 같이 저장할 수 있다.

```
>>> height = 178.9    # float 타입의 데이터를 저장한다.
```

- 수집한 데이터를 저장하기 위하여 변수를 하나씩 만들어주는 일은 번거롭기 짝이 없다. 데이터 과학자들은 많은 데이터를 처리하여야 한다. 좋은 방법이 있을까?

사람이 100명이면
변수도 100개,
사람이 1,000명이면
???????

```
>>> height1 = 178.9    # float 타입의 데이터를 저장한다.  
>>> height2 = 173.5    # float 타입의 데이터를 저장한다.  
>>> height3 = 166.1    # float 타입의 데이터를 저장한다.  
>>> height4 = 164.3    # float 타입의 데이터를 저장한다.  
>>> height5 = 176.4    # float 타입의 데이터를 저장한다.
```

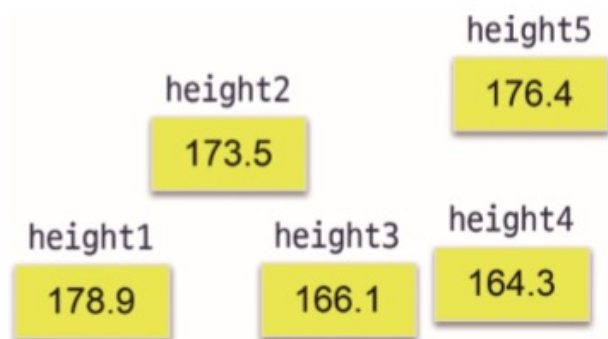


7.1 리스트는 무엇이고 왜 필요한가

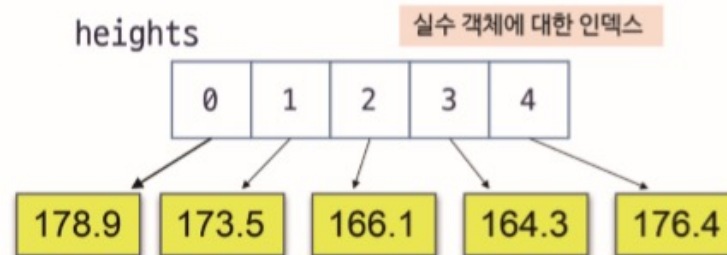
- 이럴 때는 "리스트(list)"를 만들고 여기에 데이터를 저장하면 된다. 파이썬에서 리스트는 대괄호를 이용해서 만들 수 있다. 예를 들어, 학생 5명의 키를 리스트에 저장하려면 다음과 같이 한다.

```
>>> heights = [178.9, 173.5, 166.1, 164.3, 176.4]
>>> heights
[178.9, 173.5, 166.1, 164.3, 176.4]
```

리스트를 사용하면 하나의 변수에 여러 개의 값을 담을 수 있다.



개별 변수로 표현된 데이터



리스트로 표현된 데이터

리스트 내의 개별 데이터를 항목 또는 요소라고 한다.

7.2 여러 개의 항목이 들어가는 리스트를 만들자

- 파이썬에서 리스트는 다른 변수처럼 생성할 수 있다. 즉 항목들을 `[]` 안에 적어주고 변수에 저장하면 리스트 변수가 생성된다. 다음과 같은 아이돌 그룹 BTS의 멤버 3명으로 리스트를 간단하게 만들어 보자.

```
>>> bts = ['V', 'Jungkook', 'Jimin']
```

- 만일 초기에 멤버가 없을 경우 다음과 같은 방식도 가능하다. 이 때는 빈 리스트가 만들어 진다.

```
>>> bts = []
```

7.2 여러 개의 항목이 들어가는 리스트를 만들자

- 항목이 없는 공백 리스트는 도대체 어디에 쓸모가 있을까? 공백 리스트에는 코드로 항목을 추가할 수 있다. 많은 경우에 우리는 리스트에 몇 개의 항목이 들어갈 것인지를 예측할 수 없다. 이런 경우에 공백 리스트가 유용하다. 리스트에 항목을 새롭게 추가하려면 `append(항목)` 메소드를 사용한다.

```
>>> bts = []  
>>> bts.append("V")  
>>> bts  
['V']
```

- 리스트에 요소를 추가하는 매우 간편한 방법은 `+` 연산자를 이용하는 것이다.

```
>>> bts.append("Jin")
>>> bts.append("Suga")
>>> bts
['V', 'Jin', 'Suga']
```

리스트 내에 새 항목을 추가한다.

```
>>> bts = ['V', 'Jin', 'Suga', 'Jungkook']
>>> bts = bts + ['RM'] # 덧셈 연산자로 멤버 'RM'을 추가함
>>> bts
['V', 'Jin', 'Suga', 'Jungkook', 'RM']
```

리스트 내에 새 항목 'RM'을 추가한다.

```
>>> list(range(5)) # 0에서 5사이의 정수열을 생성(5는 포함안됨)
[0, 1, 2, 3, 4]
>>> list(range(0, 5)) # list(range(5))와 동일한 결과
[0, 1, 2, 3, 4]
>>> list(range(0, 5, 1)) # list(range(0, 5))와 동일한 결과
[0, 1, 2, 3, 4]
>>> list(range(0, 5, 2)) # 생성하는 값을 2씩 증가시킴
[0, 2, 4]
>>> list(range(2, 5)) # 2에서 5-1까지의 연속된 수 2, 3, 4를 생성
[2, 3, 4]
```

range() 함수를 사용해서 여러 항목을 한꺼번에 생성할 수 있다.

7.3 리스트 연산을 해보자

- 파이썬에서는 리스트에 대해서도 다양한 연산이 가능하다. 예를 들어서 우리는 덧셈 연산으로 다음과 같이 두 개의 리스트를 합칠 수 있다.

```
>>> bts = ['V', 'J-Hope'] + ['RM', 'Jungkook', 'Jin']  
>>> bts  
['V', 'J-Hope', 'RM', 'Jungkook', 'Jin']
```

- 정수 리스트를 서로 더했을 때는 대응되는 원소끼리 더해지는 것이 아니라 리스트가 합쳐진다는 것을 명심하자. 대응되는 원소끼리 더하려면 뒤에서 설명하는 넘파이(NumPy)를 사용하여야 한다.

```
>>> mystery = [0, 1, 2] * 3    # [0, 1, 2]가 3회 반복되어 저장됨  
>>> mystery  
[0, 1, 2, 0, 1, 2, 0, 1, 2]
```

두 리스트를 합치는 연산

```
>>> numbers = [10, 20, 30] + [40, 50, 60]
>>> numbers
[10, 20, 30, 40, 50, 60]
```



```
>>> bts = ['V', 'J-Hope', 'RM', 'Jungkook', 'Jin', 'Jimin', 'Suga']
>>> 'V' in bts
True
>>> 'V' not in bts
False
```

'V'라는 멤버가 bts에 있는
가?



도전문제 7.1

대한민국의 걸그룹 '블랙핑크'의 네 명의 멤버로 구성된 black_pink 라는 리스트를 만들도록 하자.

- 1) 'Jennie'가 '블랙핑크'의 멤버인지를 in 연산을 통해서 확인해 보고 그 결과를 출력해 보도록 하자.
- 2) 'Psy'가 '블랙핑크'의 멤버인지를 in 연산을 통해서 확인해 보고 그 결과를 출력해 보도록 하자(이미지 출처 : seekpng.com).



LAB⁷⁻¹ 입력을 받아 맛있는 과일의 리스트를 만들어 보자

빈 리스트를 생성한 다음 사용자로부터 제일 좋아하는 3개의 과일을 입력받아서 리스트에 저장한다. 사용자로부터 과일을 입력받아서 리스트에 그 과일이 포함되어 있으면 '이 과일은 당신이 좋아하는 과일입니다.'를 출력하고, 그렇지 않으면 '이 과일은 당신이 좋아하는 과일이 아닙니다.'를 출력하는 프로그램을 작성한다.



원하는 결과

좋아하는 과일 이름을 입력하시오: 사과
좋아하는 과일 이름을 입력하시오: 키위
좋아하는 과일 이름을 입력하시오: 바나나
과일의 이름을 입력하세요: 바나나
이 과일은 당신이 좋아하는 과일입니다.

LAB7-1 입력을 받아 맛있는 과일의 리스트를 만들어 보자

```
fruits = []

name = input('좋아하는 과일의 이름을 입력하시오: ')
fruits.append(name)
name = input('좋아하는 과일의 이름을 입력하시오: ')
fruits.append(name)
name = input('좋아하는 과일의 이름을 입력하시오: ')
fruits.append(name)

name = input('과일의 이름을 입력하세요: ')
if name in fruits:
    print('이 과일은 당신이 좋아하는 과일입니다.')
else:
    print('이 과일은 당신이 좋아하는 과일이 아닙니다.')
```


7.4 리스트에 사용 가능한 함수를 알아보자

- 그다지 권장되지는 않지만 필요에 따라서 하나의 리스트 안에 여러 가지 타입의 데이터를 섞어서 저장하는 것도 가능하다. 예를 들어서 하나의 리스트 안에 학생들의 이름과 키를 다음과 같이 저장해 보자.

```
>>> slist1 = ['Kim', 178.9, 'Park', 173.5, 'Lee', 176.1]
>>> slist1
['Kim', 178.9, 'Park', 173.5, 'Lee', 176.1]
>>> slist1[0], slist1[2], slist1[4]
('Kim', 'Park', 'Lee')
```

- 자료형을 알아보려면 `type()` 함수를 사용한다. `slist1`에 `type()`을 적용하면 'list'라고 출력된다.

```
>>> type(slist1)
<class 'list'>
```

7.4 리스트에 사용 가능한 함수를 알아보자

학생들의 이름과 키를 묶어서 리스트로 만들고 이것들을 모아서 하나의 리스트로 만들 수도 있다

```
>>> slist2 = [ ['Kim', 178.9], ['Park', 173.5], ['Lee', 176.1] ]  
>>> slist2  
[ ['Kim', 178.9], ['Park', 173.5], ['Lee', 176.1] ]
```

- slist2는 3개의 리스트를 포함하고 있다. 자료형을 알아보려면 type() 함수를 사용한다. slist2에 type()을 적용하면 'list'라고 출력된다. 파이썬에서는 자료형에 따라서 작업 방식이 달라진다는 것은 항상 기억하자.

```
>>> type(slist2)  
<class 'list'>
```

7.4 리스트에 사용 가능한 함수를 알아보자

- 리스트에 다음과 같은 수치값이 있을 경우 `len()`, `max()`, `min()`, `sum()`, `any()` 등의 내장함수를 사용하여 편리하게 활용할 수 있다. `len()`은 리스트의 길이를 반환하며, `max()`는 리스트 항목 중 최댓값, `min()`은 리스트 항목중 최솟값을 반환한다.
- `list(range())`를 통해서 `range()` 함수가 생성한 값을 리스트에 넣을 수 있다. `any()` 함수는 리스트에 0이 아닌 원소가 하나라도 있을 경우 `True`를 반환한다. 따라서 0과 ""(공백문자열만으로 이루어진 리스트)는 `any()` 함수가 `False`를 반환한다.

```
>>> n_list = [200, 700, 500, 300, 400]
>>> len(n_list)          # 리스트의 길이를 반환한다
5
>>> max(n_list)          # 리스트 항목중 최댓값을 반환한다
700
>>> min(n_list)          # 리스트 항목중 최솟값을 반환한다
200
>>> sum(n_list)           # 리스트 항목의 합을 반환한다
2100
```

```
>>> sum(n_list) / len(n_list) # 전체의 합을 원소의 개수로 나누면 평균값을 얻을 수 있다
420.0
>>> list(range(1, 11)) # 1에서 10까지의 수를 생성하여 리스트에 넣는다
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> a_list = [0, ''] # 임의의 리스트를 생성
>>> any(n_list)       # n_list에 0이 아닌 원소가 하나라도 있는가
True
>>> any(a_list)        # a_list에 0이 아닌 원소가 하나라도 있는가
False
```


LAB7-2 리스트에 도시의 인구를 저장해보자

서울, 부산, 인천의 인구를 가지고 있는 리스트 `city_pop`을 생성해보자. 각 도시들의 인구는 `Seoul`, `Busan`, `Incheon` 등의 변수에 저장되어 있다고 가정한다. 우리는 변수를 사용해서 다음과 같이 리스트를 생성할 수 있다.

```
# 인구 통계(단위: 천명)
Seoul = 9765
Busan = 3441
Incheon = 2954
city_pop = [ Seoul, Busan, Incheon ] # 변수들로 리스트 생성
print(city_pop)                     # 리스트 데이터를 출력
```

세 도시에 대전(Daejeon)을 추가해 보도록 하자. 대전의 인구는 1,531천명이라고 가정하자. 그리고 이 네 도시중에서 가장 인구가 많은 도시의 인구와 가장 인구가 적은 도시의 인구, 그리고 네 도시의 인구의 평균을 출력해 보도록 하자. 이를 위하여 `min()`, `max()` 함수를 사용하는 대신 `for` 반복문을 사용하는 방법으로 구현해 보아라.

원하는 결과

```
[9765, 3441, 2954]
최대 인구: 9765
최소 인구: 1531
평균 인구: 4422.75
```

LAB7-2 리스트에 도시의 인구를 저장해보자

<문제에서 제시된 코드를 여기에 삽입한다>

```
Daejeon = 1531
city_pop.append(Daejeon)

max_pop = 0
min_pop = 1000000
pop_sum = 0
n = 0

for pop in city_pop:      # 순환문을 돌면서 최댓값, 최솟값을 구한다
    if pop > max_pop :
        max_pop = pop
    if pop < min_pop :
        min_pop = pop
    pop_sum += pop
    n += 1

print('최대 인구:', max_pop)
print('최소 인구:', min_pop)
print('평균 인구:', pop_sum / n)
```

7.5 인덱스를 사용하여 리스트의 항목에 접근하자

- 파이썬 리스트에 저장된 데이터를 꺼내려면 어떻게 하면 될까? 예를 들어서 다음과 같이 알파벳 문자를 저장하고 있는 리스트가 있다고 하자.

```
>>> letters = ['A', 'B', 'C', 'D', 'E', 'F']
```

- 리스트에서 데이터를 추출하려면 **인덱스index**라는 정수를 사용한다. 리스트의 첫 번째 데이터는 인덱스 0을, 두 번째 요소는 인덱스 1을 가지게 된다. 나머지 요소들도 유사하게 인덱스가 할당된다.

```
>>> letters[0] # 리스트의 첫 항목에 접근  
'A'
```

letters	'A'	'B'	'C'	'D'	'E'	'F'
인덱스	0	1	2	3	4	5



7.5 인덱스를 사용하여 리스트의 항목에 접근하자

```
>>> letters[1]  
'B'  
>>> letters[2]  
'C'
```

파이썬에서는 인덱스를
음수로 줄 수도 있다!
리스트의 맨 끝에 있는 데이
터를 얻고자 할 때 유용

```
>>> letters[-1] # 리스트의 마지막 항목에 접근  
'F'
```

letters	'A'	'B'	'C'	'D'	'E'	'F'
인덱스	0	1	2	3	4	5
음수 인덱스	-6	-5	-4	-3	-2	-1



letters[-1]

7.5 인덱스를 사용하여 리스트의 항목에 접근하자

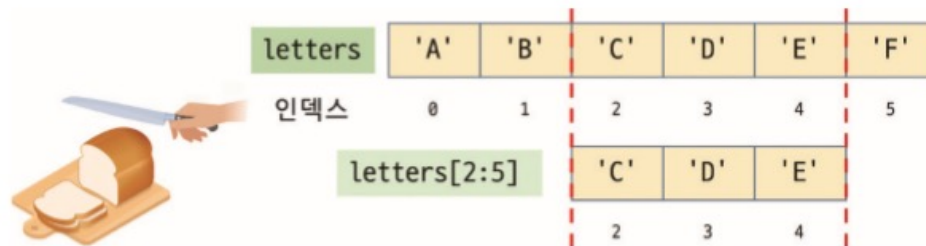
- 다음은 BTS 멤버의 영문 이름으로 이루어진 리스트이다. 이 예제는 리스트에 대하여 인덱싱을 이용하여 항목을 접근하거나 `min()`, `max()` 등의 함수를 통해서 항목을 가져오는 기능이 실려있다.

```
>>> bts = ['V', 'RM', 'Jungkook', 'J-Hop', 'Suga', 'Jin', 'Jimin']
>>> len(bts)           # 리스트의 원소 개수를 얻는다.
7
>>> bts[len(bts)-1]    # 마지막 원소
'Jimin'
>>> bts[-1]           # 음수 인덱스로 마지막 원소를 쉽게 접근할 수 있다.
'Jimin'
>>> min(bts)           # 리스트 원소 중 가장 작은 값을 찾는다. (문자열은 사전식 순서)
'J-Hop'
>>> max(bts)           # 리스트 원소 중 가장 큰 값을 찾는다. (문자열은 사전식 순서)
'V'
```

7.6 리스트를 원하는 모양으로 자르는 슬라이싱

- 리스트에서 여러 요소를 선택해서 새로운 리스트를 만들고 싶다면, 슬라이싱(slicing)이라고 하는 기능을 사용할 수 있다. 리스트를 슬라이싱 하려면 다음과 같이 콜론을 이용하여 범위를 지정한다.

```
>>> letters = ['A', 'B', 'C', 'D', 'E', 'F']  
>>> letters[2:5]  
['C', 'D', 'E']
```



- 슬라이싱을 할 때 letters[2:5]와 같이 적으면 2부터 시작하여서 항목들을 추출하다가 5가 나오기 전에 중지한다.

7.6 리스트를 원하는 모양으로 자르는 슬라이싱

- 리스트를 슬라이싱하면 원래의 리스트는 손상되지 않으며, 새로운 리스트가 생성되어서 우리에게 반환된다. 즉 슬라이스는 원래의 리스트의 부분 복사본이다.

```
>>> letters[:3]  
['A', 'B', 'C']
```

- 또한 두 번째 인덱스가 생략되면 리스트의 끝까지라고 가정한다. 이것은 생각보다 편리한 기능이다.

```
>>> letters[3:]  
['D', 'E', 'F']
```

- 극단적인 표현도 가능하다. 콜론만 있으면 리스트의 처음부터 끝까지라고 생각한다.

7.6 리스트를 원하는 모양으로 자르는 슬라이싱

```
>>> letters[:]
['A', 'B', 'C', 'D', 'E', 'F']
```

콜론만 있으면 리스트의 처음부터 끝까지

```
>>> letters[::2]
['A', 'C', 'E']
>>> letters[::-1]
['F', 'E', 'D', 'C', 'B', 'A']
```

Step 만큼 건너뛰기

뒤에서부터 앞으로 읽어오며 원소를 생성한다

7.6 리스트를 원하는 모양으로 자르는 슬라이싱



도전문제 7.2

- 1) `numbers = [4, 5, 6, 7, 8, 9]`와 같은 리스트가 있다고 하자. 6을 추출하는 문장을 적어보자.
- 2) 위의 리스트에서 슬라이싱을 통해 `[5, 6]` 리스트를 추출하도록 하자.
- 3) 시작인덱스를 생략하여 `[4, 5, 6]` 리스트를 추출하도록 하자.
- 4) 시작인덱스를 1로하고 스텝값을 2로 하여 `[5, 7, 9]`를 추출하도록 하자.

LAB⁷⁻³ 도시의 인구 자료에 대한 슬라이싱을 해보자

서울, 부산, 인천의 인구를 가지고 있는 리스트 `population`이 있다고 하자.

다음과 같은 리스트가 생성되어 있다.

```
population = ["Seoul", 9765, "Busan", 3441, "Incheon", 2954]
```

- 1) 이 리스트에서 서울의 인구인 두 번째 요소를 출력해보자.
- 2) 이 리스트에서 마지막 요소인 인천의 인구를 출력해보자. 이때, 음수로 된 인덱스를 사용해본다.
- 3) 각 도시의 이름을 `step` 값을 이용하여 출력해 보자.
- 4) 각 도시의 인구를 `step` 값을 이용하여 출력한 후 이 값들의 합을 출력하도록 하자.

원하는 결과

서울 인구: 9765

인천 인구: 2954

도시 리스트: ['Seoul', 'Busan', 'Incheon']

인구의 합: 16160

LAB⁷⁻³ 도시의 인구 자료에 대한 슬라이싱을 해보자

```
# 다음과 같은 리스트가 생성되어 있다.
population = ["Seoul", 9765, "Busan", 3441, "Incheon", 2954]

print('서울 인구:', population[1])      # 문제 1)
print('인천 인구:', population[-1])     # 문제 2)

cities = population[0::2]                # 문제 3)
print('도시 리스트:', cities)

pops = population[1::2]                  # 문제 4)
print('인구의 합:', sum(pops))
```



도전문제 7.3

- 1) `s = "Python is strong"`과 같은 문자열이 있다고 하자. 이 문자열에서 가장 첫 알파벳 문자 'P'를 추출하여라.
- 2) 위의 문자열 `s`에서 슬라이싱을 통해 'strong' 문자열을 추출하도록 하자.
- 3) 시작 인덱스를 생략한 슬라이싱을 통해 'Python' 문자열을 추출하도록 하자.
- 4) 슬라이싱을 통해 'is' 문자열을 추출하여라.

7.7 리스트의 원소 값을 자유롭게 조작해 보자

- 우리는 리스트의 요소들을 조작할 수 있다. 리스트 요소 변경은 간단하다. 원하는 요소를 인덱스를 이용하여 지정한 후에 할당연산자로 새 값을 할당하면 된다.

```
>>> slist = ['Kim', 178.9, 'Park', 173.5, 'Lee', 176.1]
```

```
>>> slist[3] = 175.0
```

'Park'의 키를 175.0로 변경하고 싶으면
인덱스를 사용하여 다음과 같이 수정

```
>>> slist  
['Kim', 178.9, 'Park', 175.0, 'Lee', 176.1]
```

7.7 리스트의 원소 값을 자유롭게 조작해 보자

- 리스트의 슬라이싱을 사용하여 한 번에 여러 원소의 값을 변경할 수도 있다. 예를 들어서 Park의 이름을 'Paik'으로, 키를 180.0으로 바꿔주고 싶은 경우, `slist[2:4]`를 써준 뒤 이름과 키를 리스트로 적으면 된다.

```
>>> slist[2:4] = ['Paik', 180.0]
>>> slist
['Kim', 178.9, 'Paik', 180.0, 'Lee', 176.1]
```

7.7 리스트의 원소 값을 자유롭게 조작해 보자

- 함수를 사용하여 동적으로 항목을 추가하려면 `append()`나 `insert()` 함수를 사용할 수 있다. `append()`는 리스트의 뒤쪽에만 추가할 수 있으나 `insert(index, item)`는 지정된 `index` 위치에 항목을 추가한다.

```
>>> slist.insert(4, "Hong")
>>> slist.insert(5, 168.1)
>>> slist
['Kim', 178.9, 'Paik', 180.0, 'Hong', 168.1, 'Lee', 176.1,]
```

함수를 사용하여 동적으로 항목을 추가하려면
`append()`나 `insert()` 함수를 사용할 수 있다

7.7 리스트의 원소 값을 자유롭게 조작해 보자

- 리스트의 메소드와 하는 일을 정리하면 다음 표와 같다.

메소드	하는 일
<code>index(x)</code>	원소 <code>x</code> 를 이용하여 위치를 찾는 기능을 한다.
<code>append(x)</code>	원소 <code>x</code> 를 리스트의 끝에 추가한다.
<code>count(x)</code>	리스트 내에서 <code>x</code> 원소의 개수를 반환한다.
<code>extend([x1, x2])</code>	<code>[x1, x2]</code> 리스트를 기존 리스트에 삽입한다.
<code>insert(index, x)</code>	원하는 <code>index</code> 위치에 <code>x</code> 를 추가한다.
<code>remove(x)</code>	<code>x</code> 원소를 리스트에서 삭제한다.
<code>pop(index)</code>	<code>index</code> 위치의 원소를 삭제한 후 반환한다. 이때 <code>index</code> 는 생략될 수 있으며 이 경우 리스트의 마지막 원소를 삭제하고 이를 반환한다.
<code>sort()</code>	값을 오름차순 순서대로 정렬한다. 키워드 인자 <code>reverse=True</code> 이면 내림차순으로 정렬한다.
<code>reverse()</code>	리스트를 원래 원소들의 역순으로 만들어 준다.

7.8 항목을 삭제하는 방법은 여러 가지가 있다

- 우리는 앞에서 리스트의 항목을 추가하거나 변경시키는 연산자와 메소드를 알아보았다. 리스트의 항목을 삭제하는 것도 가능할까? 물론이다. 삭제를 위해서는 `remove()`, `pop()` 메소드 그리고 `del` 명령을 사용할 수 있다.



- `remove()` 메소드는 다음과 같이 리스트에서 지정된 항목을 삭제한다.

7.8 항목을 삭제하는 방법은 여러 가지가 있다

`remove()` 메소드는 다음과 같이 리스트에서
지정된 항목을 삭제

```
>>> bts = [ "V", "J-Hope", "Suga", "Jungkook" ]  
>>> bts.remove("Jungkook")  
>>> bts  
['V', 'J-Hope', 'Suga']
```

조건이 참인 경우에만 bts 리스트에서 이 항목을
삭제하면 안전한 프로그램이 된다

```
if "Suga" in bts:  
    bts.remove("Suga")
```

7.8 항목을 삭제하는 방법은 여러 가지가 있다.

- `pop()` 역시 리스트에서 항목을 삭제하는 역할을 하는데 `remove()`와는 달리 리스트의 마지막 항목을 삭제하여 그 항목값을 반환한다.

```
>>> bts = ["V", "J-Hope", "Suga", "Jungkook"]
>>> last_member = bts.pop()      # 마지막 항목 'Jungkook'을 삭제하고 반환한다
>>> last_member                  # 삭제된 항목을 출력하자
'Jungkook'
>>> bts                          # bts 리스트에 마지막 항목이 삭제되었는가 확인하자
['V', 'J-Hope', 'Suga']
```

7.8 항목을 삭제하는 방법은 여러 가지가 있다.

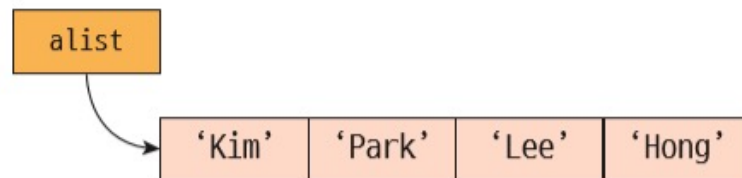
- **del** 명령어는 **리스트의 메소드가 아니므로 주의를 기울여야 한다.**
이 명령어는 파이썬의 키워드인데 다음과 같이 인덱스를 사용하여 해당하는 항목을 메모리에서 삭제할 수도 있다.

```
>>> bts = [ "V", "J-Hope", "Suga", "Jungkook"]
>>> del bts[0]                # 리스트의 첫 항목을 삭제하는 명령어
>>> bts
['J-Hope', 'Suga', 'Jungkook']
```

7.9 리스트의 객체의 생성과 참조라는 깊이있는 개념

- 우리는 파이썬 리스트가 내부적으로 어떻게 저장되고 작동하는 지를 이해하여야 한다. 왜냐하면 이것을 이해해야만 리스트를 올바르게 사용할 수 있기 때문이다. 우리가 `alist` 라는 새로운 리스트를 만들면 파이썬은 컴퓨터 메모리에 리스트를 저장한 후에, 리스트의 주소를 `alist` 변수에 저장한다.

```
>>> alist = ['Kim', 'Park', 'Lee', 'Hong']
```



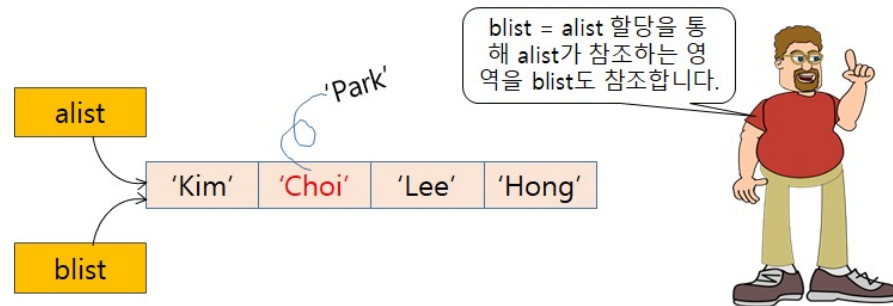
7.9 리스트의 객체의 생성과 참조라는 깊이있는 개념

- 여기서 blist의 인덱스 1의 원소를 변경하였음에도, alist 인덱스 1의 원소도 바뀌어 있는 것을 알 수 있다.

```
>>> blist = alist
>>> blist[1] = 'Choi' # blist의 두 번째 항목값을 변경함
>>> alist
['Kim', 'Choi', 'Lee', 'Hong']
```

7.9 리스트의 객체의 생성과 참조라는 깊이있는 개념

- 변수의 이름이 실제 존재하는 메모리 상의 객체를 가리키는 것을 **참조reference**라고 한다.



- `id()` 함수를 통해서 메모리에 올라온 객체의 고유한 식별번호를 알아 낼 수 있다. 이제 `id(alist)`, `id(blist)`를 통해 `alist`와 `blist`의 아이디를 조회해보면 다음과 같이 두 객체의 아이디가 같다는 것을 알 수 있다.



7.9 리스트의 객체의 생성과 참조라는 깊이있는 개념

```
▶ alist = ['Kim', 'Park', 'Lee', 'Hong']  
blist = alist  
print('alist\'s id: ', id(alist))  
print('blist\'s id: ', id(blist))  
blist[1] = 'Choi'  
print('alist: ', alist)  
print('blist: ', blist)
```

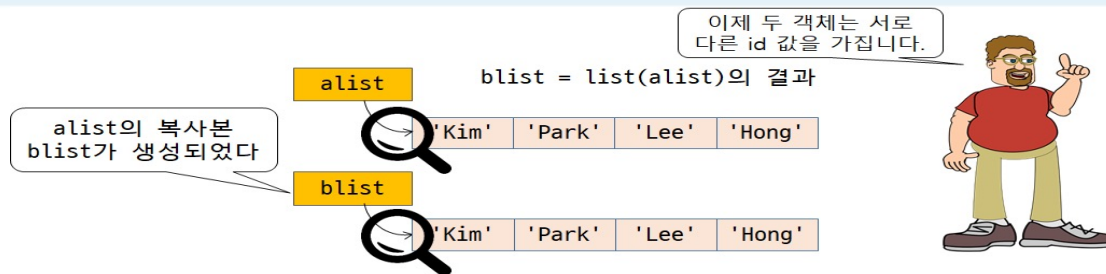
```
👤 alist's id: 139870302526192  
blist's id: 139870302526192  
alist: ['Kim', 'Choi', 'Lee', 'Hong']  
blist: ['Kim', 'Choi', 'Lee', 'Hong']
```

두 객체의 아이디가 같다!

7.10 리스트를 복제한 새로운 리스트 만들기

- 만약 동일한 내용을 가지는 새로운 리스트를 생성하고 싶은 경우에는, 단순히 등호를 써주면 안되고 'list()' 함수를 사용해야 한다. list() 함수를 사용하면 원래 리스트의 복사본이 생성되며 blist는 이 복사본을 참조한다. 따라서 두 객체의 id는 다른 값을 가진다. 이 책에서는 140050268419592라는 아이디 값이 나타나지만 이 값은 고정적이지 않으므로 여러분이 실행하면 다르게 나타날 수 있다.

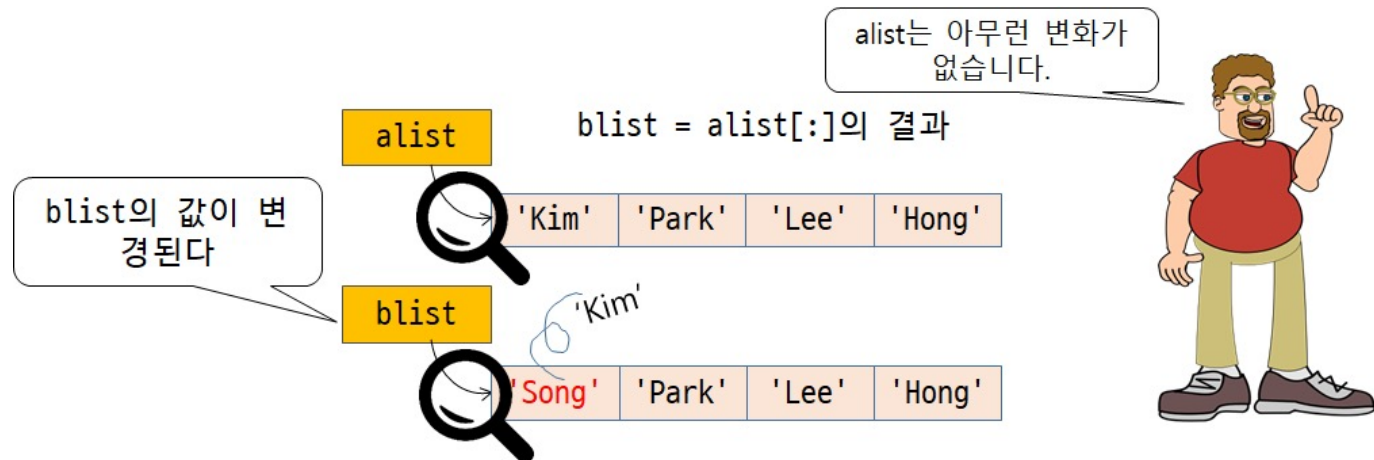
```
>>> alist = ['Kim', 'Park', 'Lee', 'Hong']
>>> blist = list(alist)
>>> id(alist)           # 책의 값과 여러분이 실행한 값은 다를 수 있다
140050268419592
>>> id(blist)           # blist의 아이디는 alist의 아이디와 다를 것이다.
140050268535624
```



```

>>> blist = alist[:]          # 이렇게 하여도 된다.
>>> id(alist)
140050268419592
>>> id(blist)                  # blist의 아이디는 alist의 아이디와 다를 것이다.
140050129233608
>>> blist[0] = 'Song'
>>> alist                      # alist의 복사본 blist를 고쳤으므로 아무런 변화가 없다
['Kim', 'Park', 'Lee', 'Hong']
>>> blist                      # alist의 복사본의 첫 번째 항목이 변경됨
['Song', 'Park', 'Lee', 'Hong']

```



7.11 리스트를 탐색해보자

- 우리는 리스트에서 특정한 항목을 찾을 수 있다. 얼마나 편리한 기능인가? C와 같은 언어에서는 이 기능을 사용하기 위해서는 상당한 코드가 있어야 한다. 반면 파이썬에서는 특정 항목이 리스트의 어떤 인덱스에 있는지를 알려면 `index()`를 사용한다.

```
>>> bts = [ "V", "J-Hope", "Suga", "Jungkook" ]  
>>> bts.index("Suga")  
2
```

- `remove()`와 마찬가지로 만약 탐색하고자 하는 항목이 리스트에 없다면 오류가 발생한다. 따라서 탐색하기 전에 `in` 연산자를 이용하여 리스트 안에 항목이 있는지 부터 확인하는 편이 안전하다.

```
if "Suga" in bts:  
    print(bts.index("Suga"))
```



7.11 리스트를 탐색해보자

- 만약 리스트의 모든 항목을 한 번씩 방문하려면 어떻게 해야 할까? 아주 간단하다. 다음과 같은 구문을 사용하면 된다. 이 방식은 반복문을 공부하며 이미 다른 바가 있다. 유용한 방법이라 리스트에서 다시 한 번 복습해 본다.

```
bts = ["V", "J-Hope", "Suga", "Jungkook" ]  
for member in bts:  
    print(member)
```

```
V  
J-Hope  
Suga  
Jungkook
```

7.12 리스트를 크기에 따라 정렬해보자

- 리스트는 정렬할 수도 있다. **정렬** `sorting`이란 리스트 안의 항목들을 크기 순으로 나열하는 것이다. 정렬을 위해서는 리스트의 `sort()` 메소드를 사용하면 된다.

```
>>> numbers = [ 9, 6, 7, 1, 8, 4, 5, 3, 2 ]
>>> numbers.sort()
>>> print(numbers)
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

- 리스트 안에 들어 있는 항목들이 문자열일 경우 사전 순서대로 정렬되는 것도 확인해 볼 수 있다.

```
>>> bts = [ "V", "J-Hope", "Suga", "Jungkook" ]
>>> bts.sort()    # bts 리스트를 알파벳 순으로 정렬
>>> print(bts)
['J-Hope', 'Jungkook', 'Suga', 'V']
```

- 만약 리스트를 역으로 정렬하고 싶으면 `sorted()`를 호출할 때, `reverse=True`을 붙인다.



```
>>> numbers.sort(reverse=True)
>>> print(numbers)
[9, 8, 7, 6, 5, 4, 3, 2, 1]
```

```
>>> numbers = [ 9, 6, 7, 1, 8, 4, 5, 3, 2 ]
>>> new_list = sorted(numbers)
>>> print(new_list)      # numbers 리스트를 항목의 크기 순으로 정렬
[1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> print(numbers)      # sorted() 함수는 정렬한 결과를 반환하므로 원래 리스트에는 변화가 없음
[9, 6, 7, 1, 8, 4, 5, 3, 2]
```

```
>>> numbers = [ 9, 6, 7, 1, 8, 4, 5, 3, 2 ]
>>> new_list = sorted(numbers, reverse=True)
>>> print(new_list)
[9, 8, 7, 6, 5, 4, 3, 2, 1]
```

7.12 리스트를 크기에 따라 정렬해보자



도전문제 7.4

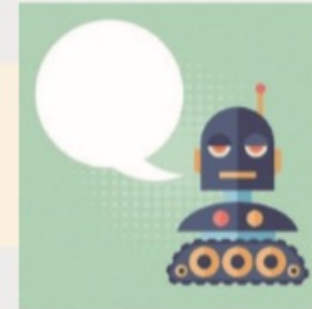
다음 코드의 실행결과를 예상해 보고 직접 실행시켜보자.

```
>>> sorted("FADEAWAY")
```

LAB7-4 오늘의 명언을 골라주는 기능을 만들자

리스트에 여러 개의 명언을 저장한 후에 그 중에서 하나를 랜덤하게 골라서 오늘의 명언으로 제공한다. 리스트에 저장한 항목 중에서 하나를 랜덤하게 고르려면 다음과 같은 random 모듈의 함수 choice()를 사용하면 간편하다.

```
import random
quotes = [...]
random.choice(quotes)
```



원하는 결과

```
#####
#           오늘의 명언           #
#####
```

고생 없이 얻을 수 있는 진실로 귀중한 것은 하나도 없다.

LAB⁷⁻⁴ 오늘의 명언을 골라주는 기능을 만들자

```
import random

quotes = []
quotes.append("꿈을 지녀라. 그러면 어려운 현실을 이길 수 있다.")
quotes.append("분노는 바보들의 가슴속에서만 살아간다..")
quotes.append("고생 없이 얻을 수 있는 진실로 귀중한 것은 하나도 없다.")
quotes.append("사람은 사랑할 때 누구나 시인이 된다.")
quotes.append("시작이 반이다.")

print("#####")
print("# 오늘의 명언 #")
print("#####")
print("")
dailyQuote = random.choice(quotes)
print(dailyQuote)
```

7.12 리스트를 크기에 따라 정렬해보자



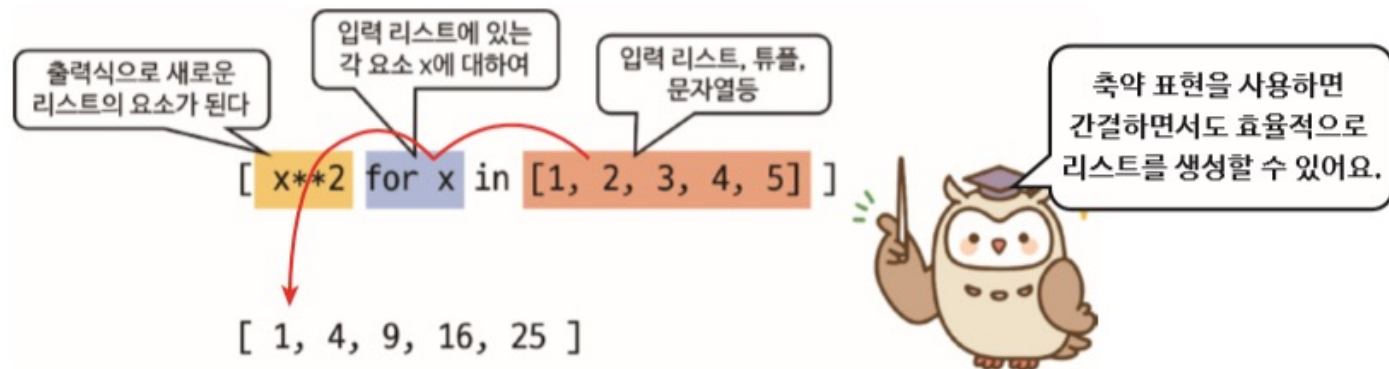
도전문제 7.5

"100 + 200"과 같은 산수 문제를 10개 저장하고 있는 리스트를 작성하고 여기에서 랜덤하게 하나를 골라서 사용자에게 제공하는 프로그램을 작성하라. 다음과 같이 `eval()` 함수를 사용하면 문자열로 이루어진 이 문장을 파이썬 수식으로 평가하여 연산을 수행하는 것이 가능하다.

```
>>> eval("100 + 200") # 문자열을 파이썬 문장으로 해석하여 100 + 200을 수행함
300
```

7.13 리스트 함축은 코드를 짧고 간결하게 만드는데 사용된다.

- 파이썬은 **리스트 함축** `list comprehension`이라는 기능을 지원한다. 영어 `comprehension`은 함축, 포함, 내포라는 의미를 가지는데, 이는 수학자들이 집합을 정의하는 것과 유사하다.
- 리스트 함축 표현을 사용하면 입력 리스트의 각 항목을 하나하나 방문하며 지정된 연산을 하고, 연산의 결과로 생성된 항목값을 가지는 리스트를 생성할 수 있다.



7.13 리스트 함축은 코드를 짧고 간결하게 만드는데 사용된다.

```
s = []  
for x in [1, 2, 3, 4, 5]:  
    s.append(x**2)
```

리스트 함축을 사용하지 않는다면
다음과 같이 번거로운 반복문을
사용해야 할 것

7.13 리스트 함축은 코드를 짧고 간결하게 만드는데 사용된다

- 리스트 함축은 다음과 같이 다양한 기능도 수행할 수 있으므로 파이썬 개발자들이 매우 빈번하게 사용하고 있다.

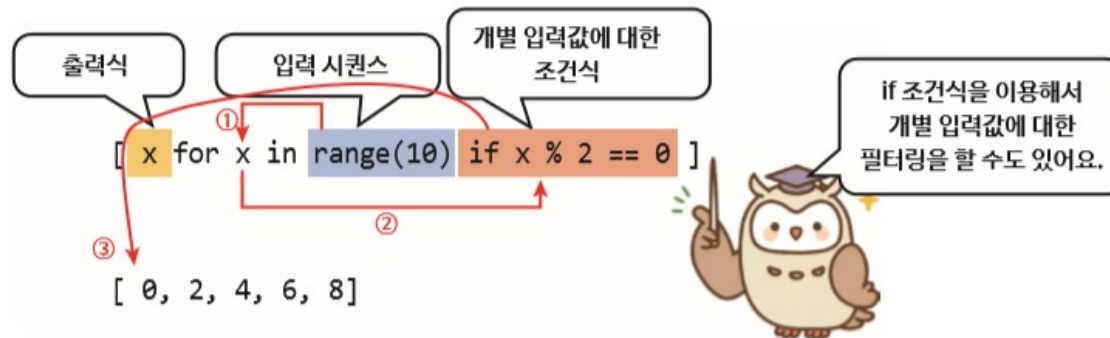
```
>>> [x for x in range(10)]      # 0에서 9까지 숫자를 포함하는 리스트
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> [x * x for x in range(10)] # 0에서 9까지 숫자의 제곱 값
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
>>> st = 'Hello World'
>>> [x.upper() for x in st]     # 문자열 각각에 대해 upper() 메소드 적용
['H', 'E', 'L', 'L', 'O', ' ', 'W', 'O', 'R', 'L', 'D']
```

- 만일 ['welcome', 'to', 'the', 'python', 'world']라는 항목을 가지는 리스트 a로부터 첫 알파벳을 추출한 다음 이를 모두 대문자로 변환한 first_a 리스트를 하기 위해서는 다음과 같은 인덱싱 기능을 사용하는 것 만으로도 가능할 것이다.

```
>>> a = ['welcome', 'to', 'the', 'python', 'world']
>>> first_a = [ s[0].upper() for s in a] # 첫 알파벳에 대한 대문자 변환
>>> print(first_a)
['W', 'T', 'T', 'P', 'W']
```

7.14 조건이 붙는 리스트 함축표현도 가능하다

- 리스트 함축에는 if를 사용하여 조건이 추가될 수 있다. 예를 들어서 0부터 9 사이의 정수 중에서 짝수의 집합을 리스트 함축으로 표현하면 다음과 같다.



```
>>> [x for x in range(10) if x % 2 == 0]
[0, 2, 4, 6, 8]
>>> [x**2 for x in range(10) if x % 2 == 0] # 출력식에 제공할 수 있다
[0, 4, 16, 66, 64]
```

- 리스트 함축은 숫자에 대해서만 적용되는 것은 아니며, 문자열과 같이 여러 자료형에 대해서도 리스트 함축을 적용할 수 있다

```
s = ["Hello", "12345", "World", "67890"]
numbers = [x for x in s if x.isdigit()]
print(numbers)
```

```
['12345', '67890']
```

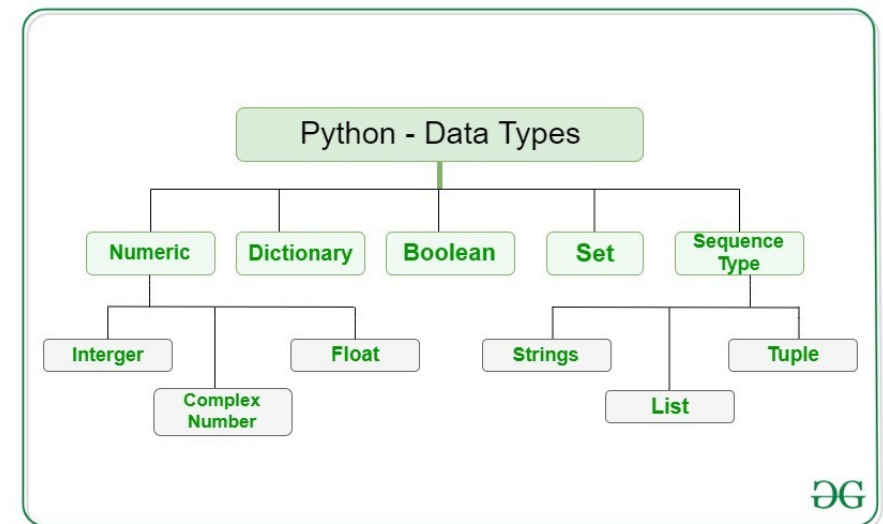
```
>>> [int(x) for x in input('정수를 여러개 입력하세요 : ').split()]
정수를 여러개 입력하세요 : 100 200 300
[100, 200, 300]
>>> [int(x) for x in input('정수를 여러개 입력하세요 : ').split() if x.isdigit()]
정수를 여러개 입력하세요 : 100 이백 300 400
[100, 300, 400]
```

```
>>> [(x, y) for x in [1, 2, 3] for y in [3, 1, 4]]
[(1, 3), (1, 1), (1, 4), (2, 3), (2, 1), (2, 4), (3, 3), (3, 1), (3, 4)]
>>> [(x, y) for x in [1, 2, 3] for y in [3, 1, 4] if x != y]
[(1, 3), (1, 4), (2, 3), (2, 1), (2, 4), (3, 1), (3, 4)]
```


7.15 한번 생성하면 그 값을 고칠 수 없는 자료형 : 튜플

- 파이썬의 데이터 형을 크게 분류하면 다음 그림과 같이 **불변속성immutable** 데이터 형, **가변속성mutable** 데이터 형으로 나눌 수 있다. **튜플tuple**은 리스트와 아주 유사하다. 하지만 튜플의 내용은 한 번 지정되면 변경될 수 없으며 대표적인 불변속성 자료형이다.
- 튜플은 한 번 그 내용이 정해지면 변경이 불가능하므로 구조가 단순하고 **리스트에 비하여 접근 속도가 빠르다**는 장점도 있다. 따라서 두 자료형은 목적에 따라서 적합한 경우에 선택하여 사용한다.

Data Structure	Ordered	Mutable	Constructor	Example
List	Yes	Yes	[] or list()	[5.7, 4, 'yes', 5.7]
Tuple	Yes	No	() or tuple()	(5.7, 4, 'yes', 5.7)
Set	No	Yes	{ }* or set()	{5.7, 4, 'yes'}
Dictionary	No	Yes**	{ } or dict()	{'Jun': 75, 'Jul': 89}



7.15 한번 생성하면 그 값을 고칠 수 없는 자료형 : 튜플

```
>>> colors = ("red", "green", "blue")  
>>> colors  
( 'red', 'green', 'blue' )
```

색상을 저장하는 튜플

```
>>> numbers = (1, 2, 3, 4, 5 )  
>>> numbers  
(1, 2, 3, 4, 5)
```

다수의 정수를 저장하는 튜플

7.15 한번 생성하면 그 값을 고칠 수 없는 자료형 : 튜플

- 다시 한번 강조하면 튜플은 **변경될 수 없는 객체** `immutable`이므로 튜플의 요소는 변경될 수 없다. 따라서 다음과 같이 변경을 시도하면 에러를 출력한다.

```
numbers = ( 1, 2, 3, 4, 5)
numbers[0] = 2
```

```
-----
TypeError                                Traceback (most recent call last)
<ipython-input-8-2bbe5c98f070> in <module>()
      1 numbers = ( 1, 2, 3, 4, 5)
----> 2 numbers[0] = 2

TypeError: 'tuple' object does not support item assignment
```

7.15 한번 생성하면 그 값을 고칠 수 없는 자료형 : 튜플

- 튜플도 시퀀스의 일종이기 때문에 인덱싱과 슬라이싱은 문자열이나 리스트와 동일하게 동작한다. 또한 튜플간의 결합연산인 +와 반복연산자 *도 사용할 수 있다

```
>>> t1 = (1, 2, 3, 4, 5)
>>> t1[0]          # 튜플의 인덱싱-리스트 인덱싱과 동일한 방식
1
>>> t1[1:4]        # 튜플의 슬라이싱 결과로 튜플을 반환함
(2, 3, 4)
>>> t2 = t1 + t1    # 튜플의 결합 연산
>>> t2
(1, 2, 3, 4, 5, 1, 2, 3, 4, 5)
```

LAB⁷⁻⁵ 함수는 튜플을 돌려줄 수 있다

C나 C++, Java와 같은 프로그래밍 언어에서는 함수가 하나의 값을 반환할 수 있다. 파이썬에서는 함수가 튜플을 반환하게 하면 함수가 여러 개의 값을 동시에 반환할 수 있다. 원의 넓이와 둘레를 동시에 반환하는 함수를 작성하고 테스트 해보자.

원하는 결과

원의 반지름을 입력하시오: 10

원의 넓이는 314.1592653589793이고 원의 둘레는 62.83185307179586이다.

LAB⁷⁻⁵ 함수는 튜플을 돌려줄 수 있다

```
import math

def calCircle(r):
    # 반지름이 r인 원의 넓이와 둘레를 동시에 반환하는 함수 (area, circum)
    area = math.pi * r * r
    circum = 2 * math.pi * r
    return area, circum    # 튜플을 반환함

radius = float(input("원의 반지름을 입력하시오: "))
(a, c) = calCircle(radius)
print("원의 넓이는 "+str(a)+"이고 원의 둘레는"+str(c)+"이다.")
```



잠깐 - 임시 변수 없는 데이터 교환

두 변수의 값을 서로 교환하여 바꾸는 일이 종종 필요하다. 초보 프로그래머가 종종 실수하는 예는 다음처럼 서로 할당을 하는 것이다.

```
a = b  
b = a
```

그러나 첫 문장 수행후 a의 값은 b로 바뀌기 때문에 두 번째 문장은 의미가 없어지고, 두 변수는 모두 원래의 b가 가지고 있던 값으로 변해 같은 값이 되어 버린다. 이를 피하기 위해 임시변수를 사용해야 한다.

```
temp = a  
a = b  
b = temp
```

그런데 튜플을 이용하면 이런 임시 변수없이 간단히 데이터를 교환할 수 있다.

```
a, b = b, a
```

LAB⁷⁻⁶ 도시의 이름과 인구를 튜플로 묶어보자

서울, 부산, 인천, 광주, 대전의 인구를 가지고 있는 리스트 `city_info`가 있다고 하자. LAB 7-3은 서로 다른 자료형인 도시명과 인구수를 하나의 리스트에 넣었다. 그러나 리스트에는 동일한 자료형을 항목으로 사용하는 것이 바람직하므로 다음과 같이 튜플의 리스트를 만들 수 있다.

다음과 같은 리스트가 생성되어 있다.

```
city_info = [('서울', 9765), ('부산', 3441), ('인천', 2954),  
             ('광주', 1501), ('대전', 1531)]
```

최대 인구를 가진 도시를 찾아 보고, 최소 인구를 가진 도시도 찾아 보라. 그리고 아래와 같이 전체 도시들의 평균 인구를 계산하는 코드도 작성해 보라.

원하는 결과

```
최대인구: 서울, 인구: 9765 천명  
최소인구: 광주, 인구: 1501 천명  
리스트 도시 평균 인구: 3838.4 천명
```


LAB7-5 도시의 이름과 인구를 튜플로 묶어보자

<문제에서 제시된 코드를 여기에 삽입한다>

```
max_pop = 0
min_pop = 999999999999999999
total_pop = 0

max_city = None
min_city = None

for city in city_info:
    total_pop += city[1]
    if city[1] > max_pop :
        max_pop = city[1]
        max_city = city
    if city[1] < min_pop :
        min_pop = city[1]
        min_city = city

print('최대인구: {0}, 인구: {1} 천명'.format(max_city[0], max_city[1]))
print('최소인구: {0}, 인구: {1} 천명'.format(min_city[0], min_city[1]))
print('리스트 도시 평균 인구: {0} 천명'.format(total_pop / len(city_info)) )
```

7.16 클래스와 객체가 무엇인가

- 프로그램 코드가 복잡해지면 이를 관리하기가 쉽지 않다. 이러한 어려움을 해결하기 위해 소프트웨어 엔지니어들은 수십년간 다양한 방법으로 해결책을 찾았으며 그 중에서 가장 뛰어난 해결책으로 인정 받고 있는 방법이 바로 **객체지향 프로그래밍** object oriented programming 방식이다
- 객체지향 프로그래밍은 다양한 객체를 **클래스** class로 미리 정의해두고 이 객체들이 프로그램 상에서 상호작용하면서 원하는 작업을 수행하는 문제해결 방식이다. 앞서 다루어 본 자료형인 리스트라는 것도 일종의 클래스이다. 우리는 이 리스트라는 클래스의 특징을 가지는 객체를 만들 수 있다. 이 각각의 객체들은 클래스의 틀을 지키면서 각자 서로 다른 실제 값을 담고 있다. 이런 객체를 **인스턴스** instance 객체라고 한다.
- 파이썬은 '객체지향 프로그래밍 언어'이다. 그리고 이미 살펴본 자료형, 함수, 모듈은 모두 객체이다.

7.16 클래스와 객체가 무엇인가

```
>>> animals = ['lion', 'tiger', 'cat', 'dog']
>>> animals.sort()      # animals 리스트의 내부 문자열을 알파벳 순으로 정렬한다
>>> animals
['cat', 'dog', 'lion', 'tiger']
>>> animals.append('rabbit') # animals 리스트에 새 원소를 추가한다
>>> animals
['cat', 'dog', 'lion', 'tiger', 'rabbit']
>>> animals.reverse()    # animals 리스트를 원래 원소의 역순으로 재배열한다
>>> animals
['rabbit', 'tiger', 'lion', 'dog', 'cat']
```

문자열 항목들을 속성으로 집어 넣어
animals라는 리스트 클래스의 인스턴스 객체를 만들었다

7.16 클래스와 객체가 무엇인가

- 이렇게 특정한 클래스에 속한 객체들이 사용할 수 있는 함수들을 해당 클래스의 **메소드** `method`라고 부른다. 객체 지향 언어는 리스트와 같은 클래스를 정의할 수 있고, 해당 클래스의 인스턴스 객체를 생성할 수 있으며, 이들이 메소드를 이용하여 다양한 일을 할 수 있도록 지원하는 언어를 의미한다.

클래스 정의

이름: 리스트

속성

복수의 데이터 나열, 데이터 개수, ...

기능 - 메소드

순서대로 정렬하기, 데이터 추가, 순서 뒤집기 등의 기능

`sort()`, `append()`, `reverse()`

프로그램에서 변수에 할당되어 동작하는 객체

리스트 클래스에 속하는 인스턴스 객체

animals

'lion' 'tiger' 'cat' 'dog'

fruits

'pear' 'apple' 'banana'

odd_numbers

1 3 5 7 9

메소드를 사용했을 때 속성의 변화

`animals.reverse()`

animals 'dog' 'cat' 'tiger' 'lion'

`fruits.sort()`

fruits 'apple' 'banana' 'pear'

`odd_numbers.append(11)`

odd_numbers 1 3 5 7 9 11



summary

핵심 정리



- 리스트는 항목들을 모아둔 곳이다.
- 리스트의 항목은 어떤 것이든 가능하다.
- 공백 리스트를 만들고 `append()`를 호출하여서 코드로 항목을 추가할 수 있다.
- 튜플은 변경불가능 자료형으로 리스트에 비하여 처리속도가 빠르지만 그 항목값을 수정할 수 없다.
- 튜플을 사용하여 함수로부터 여러 개의 값을 반환 받을 수 있다.
- 파이썬은 여러가지 종류의 클래스를 미리 만들어서 개발시에 사용할 수 있도록 제공하는 객체지향 프로그래밍 언어이다.

따라하며 배우는

파이썬과 데이터 과학



Questions?