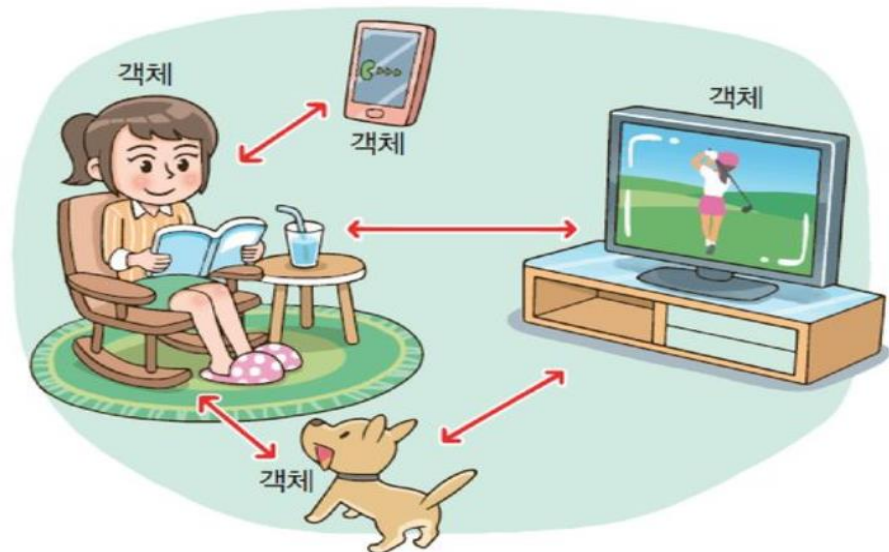


객체(object)

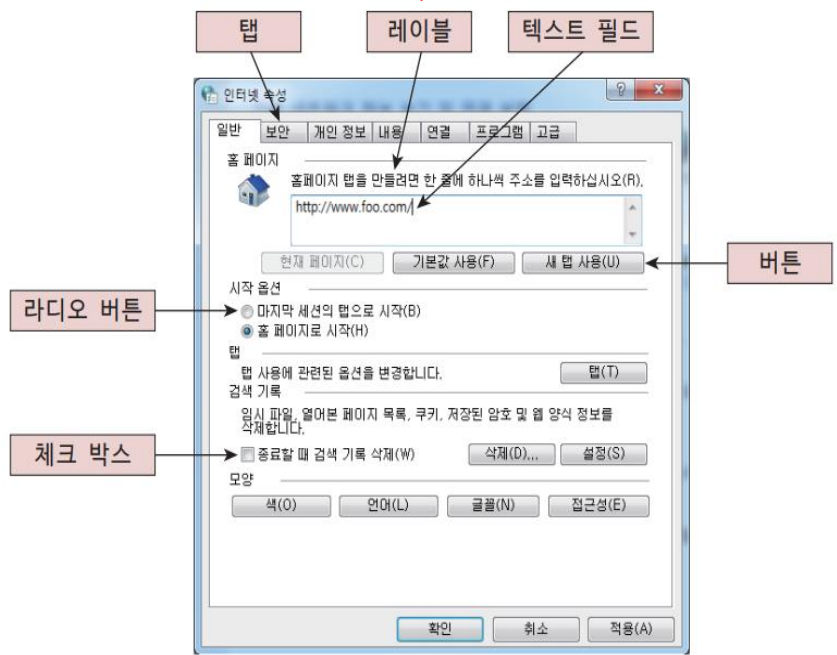
❖ 객체 지향 프로그래밍 OOP: object oriented programming

- 프로그램을 잘 때, 프로그램을 실제 세상에 가깝게 모델링 하는 기법
- 컴퓨터가 수행하는 작업을 **객체들 사이의 상호작용**으로 표현
- 클래스 **class**나 객체 **object** 들의 집합으로 소프트웨어를 개발하자는 개념
- Java, Python, C++, C#, Swift 등 현재 사용중인 많은 프로그래밍 언어에서 채택



객체(object)

아래의 다양한 구성요소(객체)들이 상호작용하도록 하자는 것이 객체지향 프로그래밍의 핵심



[그림 9-2] 윈도 운영체제 인터넷 속성창의 나타난 여러 가지 그래픽 객체들



[그림 9-3] 게임 화면과 객체들의 상호 작용

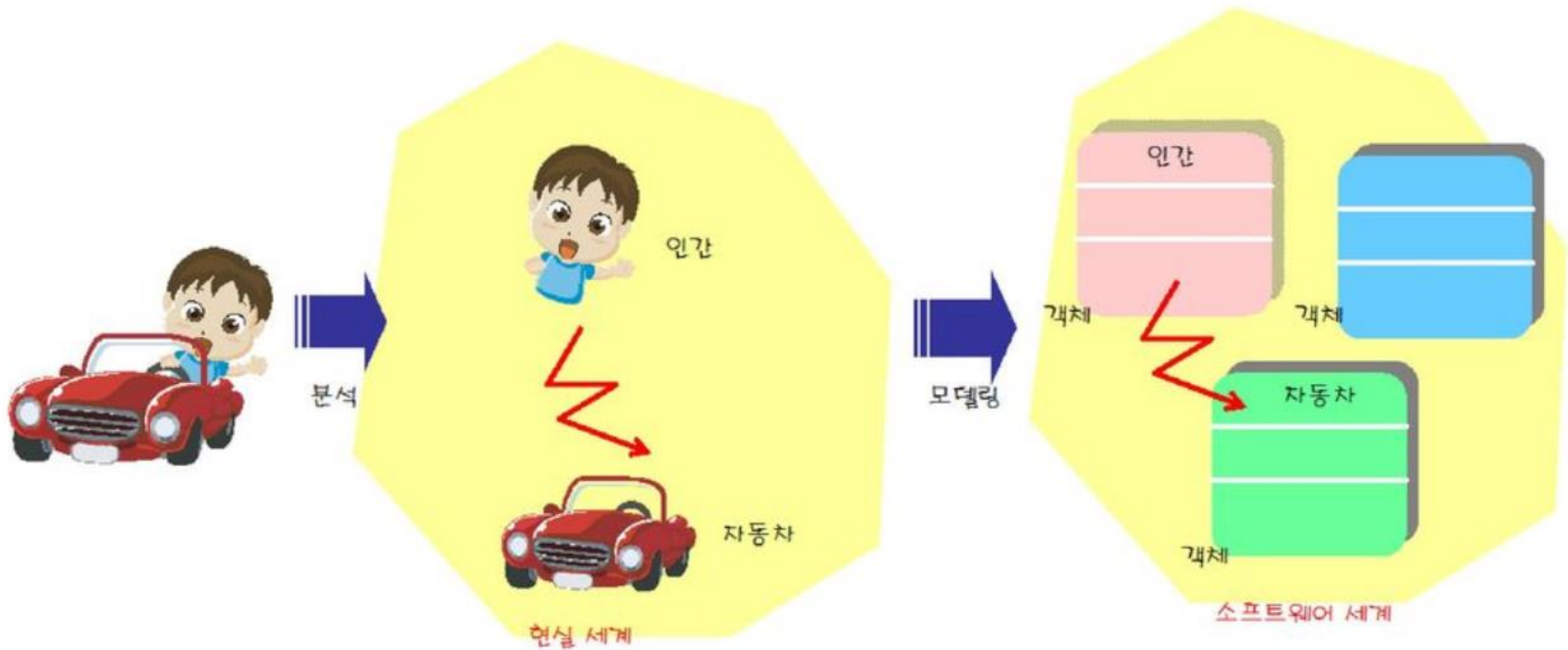
객체(object)

❖ 파이썬은 객체지향 프로그래밍(OOP) 언어



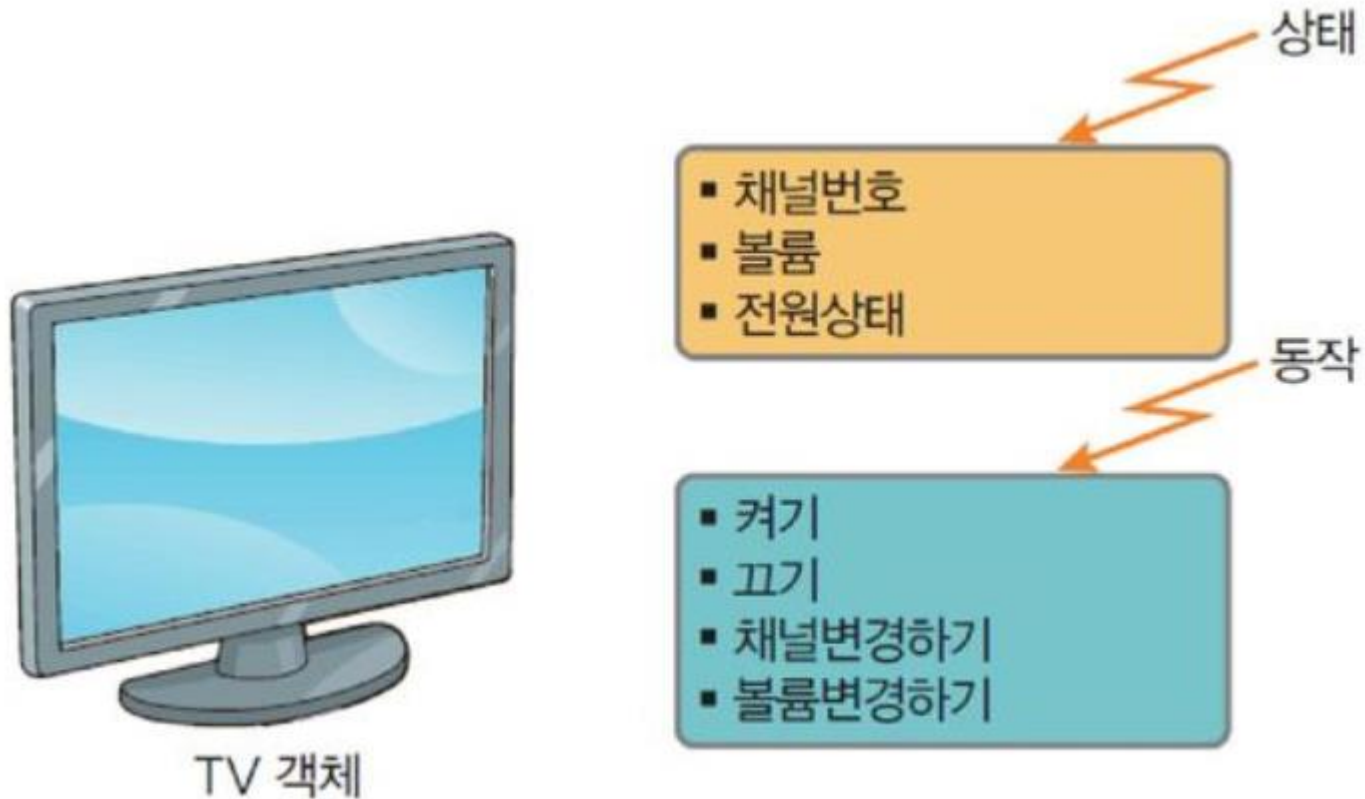
객체(object)

❖ 파이썬은 객체지향 프로그래밍(OOP) 언어



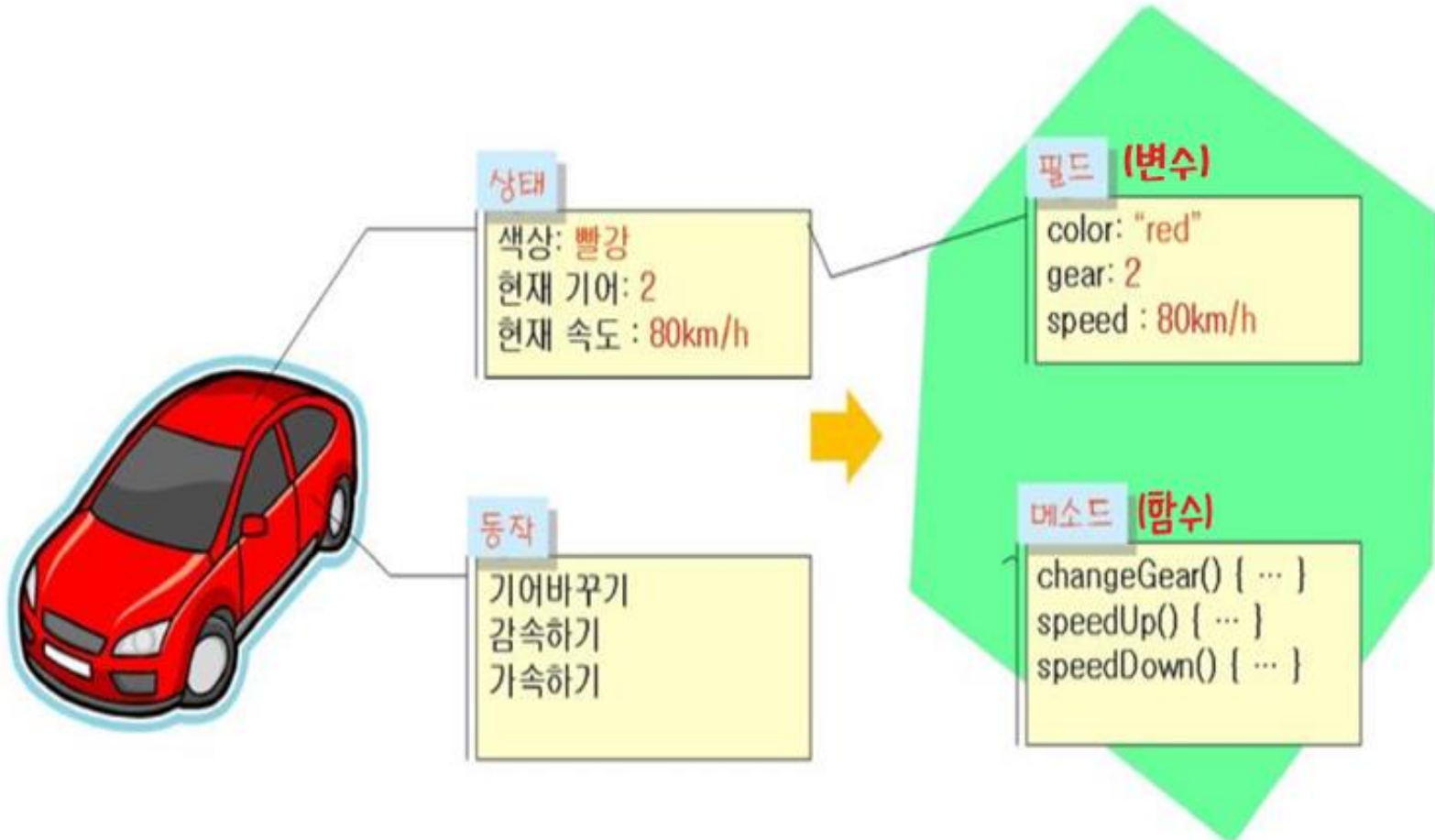
객체(object)

❖ 파이썬은 객체지향 프로그래밍(OOP) 언어



객체(object)

❖ 파이썬은 객체지향 프로그래밍(OOP) 언어

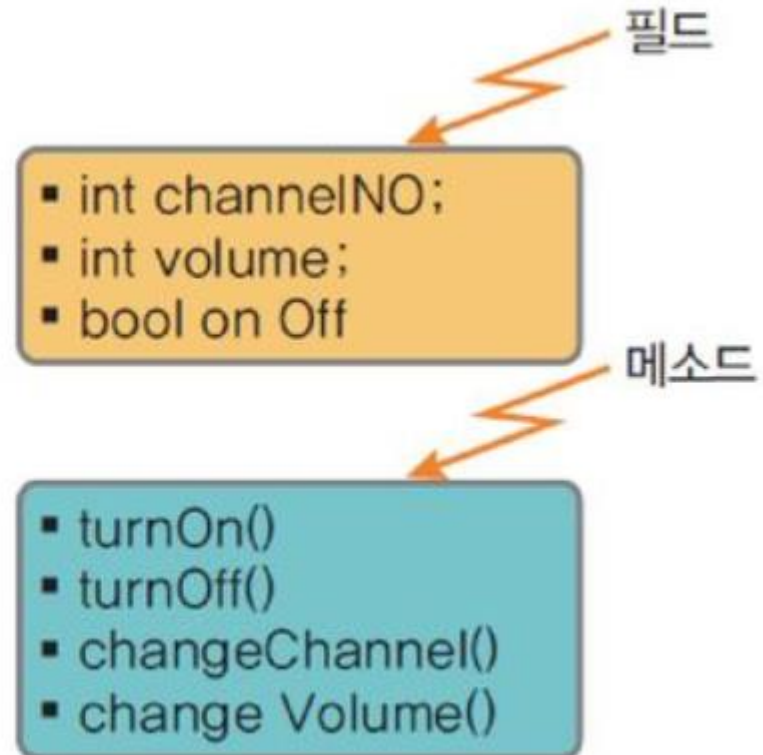


객체(object)

❖ 파이썬은 객체지향 프로그래밍(OOP) 언어

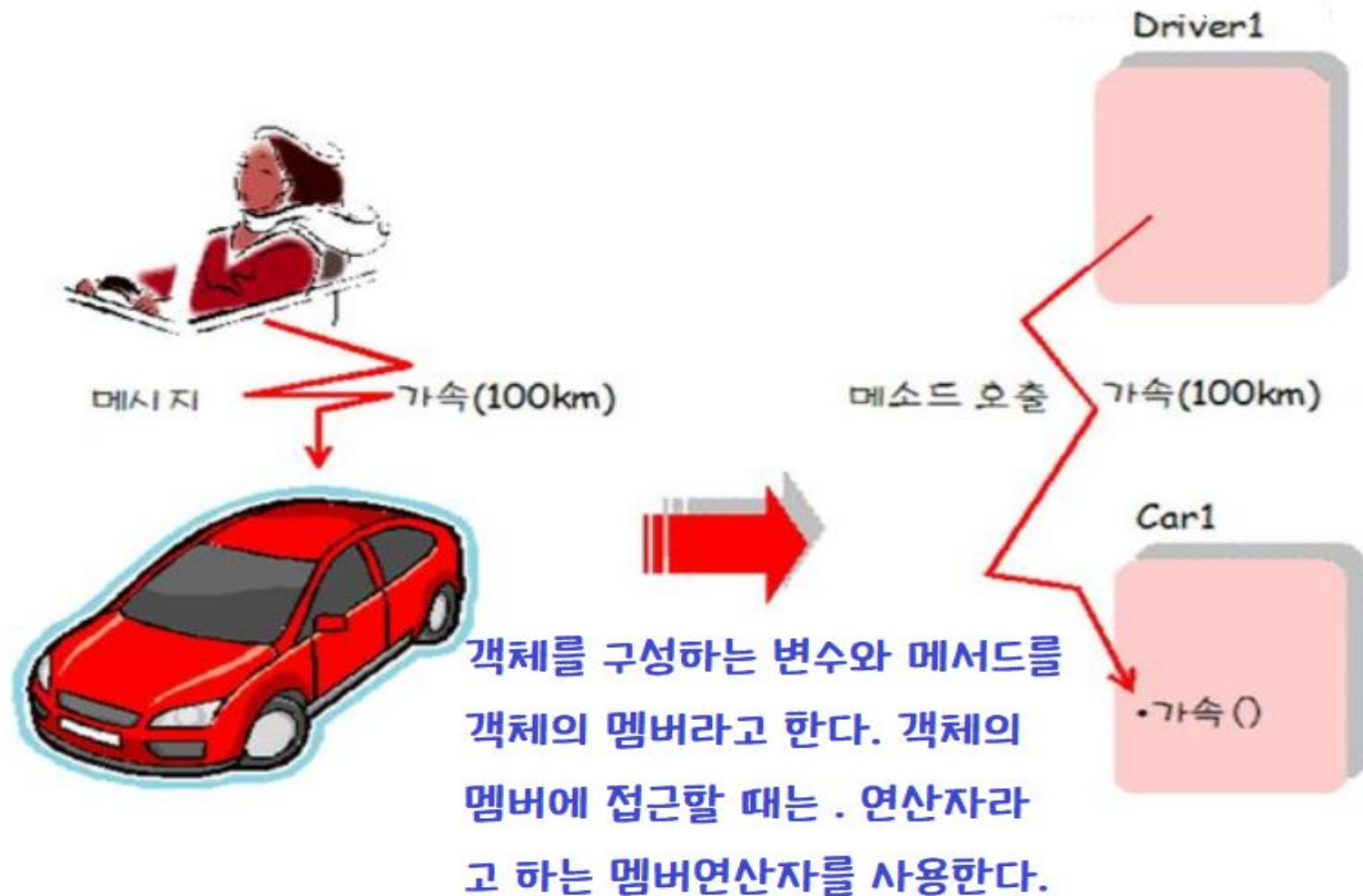


TV 객체



객체(object)

❖ 파이썬은 객체지향 프로그래밍(OOP) 언어



객체(object)

❖ 파이썬은 객체지향 프로그래밍(OOP) 언어

객체(Object) = 속성(Attribute) + 기능(Method)

= 변수(Field) + 함수(Function)

지금까지 학습한 문자열, 리스트, 튜플, 딕셔너리, 집합은 모두 콜렉션 객체들이다.

. 연산자를 사용하여 각 객체가 가지고 있는 함수(메서드) 호출이 가능하다.

"abc".upper(), [4,15,2,30,4].count(4), {1:100, 2:88, 3:90}.get(3)

객체(object)

❖ 파이썬은 객체지향 프로그래밍(OOP) 언어

다른 객체의 역할이 필요할 때 다음 형식으로 다른 객체의 메서드 또는 변수를 사용

객체 또는 객체를 담고 있는 변수

.

멤버

객체에 속한 함수를
메서드라고 한다.



❖ 문자열

Base Type + Container Type + Value + Object

```
my_string1 = "apple"
```

```
my_string2 = 'Hello, world!'
```

1. 문자열 분리

❖ 첨자

- 문자의 위치
- 대괄호와 첨자 적어 문자열 구성하는 개별 문자 읽음
- 앞뒤 양쪽에서 읽을 수 있음

0	1	2	3	4	5
p	y	t	h	o	n
-6	-5	-4	-3	-2	-1

index

```
s = "python"
print(s[2])
print(s[-2])
```

실행결과

t
o

- for

stringindex

```
s = "python"
for c in s:
    print(c, end = ',')
```

실행결과

p,y,t,h,o,n,

1. 문자열 분리

❖ 슬라이스

- 범위 지정하여 부분 문자열을 추출
- `[begin:end:step]`
 - 시작, 끝, 중간값을 지정

slice

```
s = "python"
print(s[2:5])
print(s[3:])
print(s[:4])
print(s[2:-2])
```

실행결과

```
tho
hon
pyth
th
```

1. 문자열 분리

- 일정 형식 가진 문자열에서 원하는 정보만 추출할 수 있음

slice2

```
file = "20171224-104830.jpg"
print("촬영 날짜 : " + file[4:6] + "월 " + file[6:8] + "일")
print("촬영 시간 : " + file[9:11] + "시 " + file[11:13] + "분")
print("확장자 : " + file[-3:])
```

실행결과

```
촬영 날짜 : 12월 24일
촬영 시간 : 10시 48분
확장자 : jpg
```


2. 문자열 메서드

❖ 메서드

- 클래스에 소속된 함수
- 객체에 대해 특화된 작업 수행

❖ 검색

■ len 함수

- 문자열이 길이를 조사

find

```
s = "python programming"
print(len(s))
print(s.find('o'))
print(s.rfind('o'))
print(s.index('r'))
print(s.count('n'))
```

실행결과

```
18
4
9
8
2
```

2. 문자열 메서드

■ find 메서드

- 인수로 지정한 문자 또는 부분 문자열의 위치 조사

■ rfind 메서드

- 뒤에서 검색 시작

■ count 메서드

- 특정 문자 개수

2. 문자열 메서드

❖ 조사

■ in 구문

- 특정 문자 유무 여부 조사

in

```
s = "python programming"
print('a' in s)
print('z' in s)
print('pro' in s)
print('x' not in s)
```

실행결과

```
True
False
True
True
```

2. 문자열 메서드



함수	설명
isalpha	모든 문자가 알파벳인지 조사한다.
islower	모든 문자가 소문자인지 조사한다.
isupper	모든 문자가 대문자인지 조사한다.
isspace	모든 문자가 공백인지 조사한다.
isalnum	모든 문자가 알파벳 또는 숫자인지 조사한다.
isdecimal	모든 문자가 숫자인지 조사한다.
isdigit	모든 문자가 숫자인지 조사한다.
isnumeric	모든 문자가 숫자인지 조사한다.
isidentifier	명칭으로 쓸 수 있는 문자로만 구성되어 있는지 조사한다.
isprintable	인쇄 가능한 문자로만 구성되어 있는지 조사한다.

2. 문자열 메서드

❖ 변경

■ lower 메서드 / upper 메서드

- 각기 영문자를 전부 소문자 / 대문자로 바꿈

lower

```
s = "Good morning. my love KIM."  
print(s.lower())  
print(s.upper())  
print(s)  
  
print(s.swapcase())  
print(s.capitalize())  
print(s.title())
```

실행결과

```
good morning. my love kim.  
GOOD MORNING. MY LOVE KIM.  
Good morning. my love KIM.  
gOOD MORNING. MY LOVE kim.  
Good morning. my love kim.  
Good Morning. My Love Kim.
```

2. 문자열 메서드

- 문자열 자체를 변경하는 것은 아님

- `rstrip` / `rstrip` / `strip` 메서드

- 왼쪽 / 오른쪽 / 양측 공백을 제거

`strip`

```
s = "  angel  "
print(s + "님")
print(s.lstrip() + "님")
print(s.rstrip() + "님")
print(s.strip() + "님")
```

실행결과

```
angel  님
angel  님
angel님
angel님
```


2. 문자열 메서드

❖ 분할

■ split 메서드

- 구분자를 기준으로 문자열을 분할

split

```
s = "짜장 짬뽕 탕숙"
print(s.split())

s2 = "서울->대전->대구->부산"
city = s2.split("->")
print(city)
for c in city:
    print(c, "찍고", end = ' ')
```

실행결과

```
['짜장', '짬뽕', '탕숙']
['서울', '대전', '대구', '부산']
서울 찍고 대전 찍고 대구 찍고 부산 찍고
```

■ splitlines 메서드

- 개행 문자나 파일 구분자 등 기준으로 문자열 잘라 리스트로 만들

2. 문자열 메서드

■ join 메서드

- 문자열의 각 문자 사이에 다른 문자열 끼워넣음

join

```
s = "._."  
print(s.join("대한민국"))
```

실행결과

대._.한._.민._.국

2. 문자열 메서드

❖ 대체

■ replace 메서드

- 특정 문자열을 찾아 다른 문자열로 대체
- 첫 번째 인수 : 검색할 문자열 지정
- 두 번째 인수 : 바꿀 문자열 지정

replace

```
s = "독도는 일본땅이다. 대마도도 일본땅이다."  
print(s)  
print(s.replace("일본", "한국"))
```

실행결과

```
독도는 일본땅이다. 대마도도 일본땅이다.  
독도는 한국땅이다. 대마도도 한국땅이다.
```

문자열 분리와 결합 복습



```
>>> items = 'zero one two three'.split()           # 빈칸을 기준으로 문자열 분리하기
>>> print (items)
['zero', 'one', 'two', 'three']
```

```
>>> example = 'python,jquery,javascript'           # ","를 기준으로 문자열 나누기
>>> example.split(",")
['python', 'jquery', 'javascript']
>>> a, b, c = example.split(",")                   # 리스트에 있는 각 값을 a, b, c 변수로 언패킹
>>> print(a, b, c)
python jquery javascript
>>> example = 'theteamlab.univ.edu'
>>> subdomain, domain, tld = example.split('.')     # "."을 기준으로 문자열 나누기 → 언패킹
>>> print(subdomain, domain, tld)
theteamlab univ edu
```

문자열 분리와 결합 복습



```
>>> colors = ['red', 'blue', 'green', 'yellow']
>>> result = ''.join(colors)
>>> result
'redbluegreenyellow'
```

```
>>> result = ' '.join(colors)           # 연결 시, 1칸을 띄고 연결
>>> result
'red blue green yellow'
>>> result = ', '.join(colors)          # 연결 시 ", "으로 연결
>>> result
'red, blue, green, yellow'
>>> result = '-'.join(colors)           # 연결 시 "-"으로 연결
>>> result
'red-blue-green-yellow'
```

문자열 관련 주요 함수와 메서드



함수명	기능
len()	문자열의 문자 개수를 반환 
upper()	대문자로 변환
lower()	소문자로 변환
title()	각 단어의 앞글자만 대문자로 변환
capitalize()	첫 문자를 대문자로 변환
count('찾을 문자열')	'찾을 문자열'이 몇 개 들어 있는지 개수 반환
find('찾을 문자열')	'찾을 문자열'이 왼쪽 끝부터 시작하여 몇 번째에 있는지 반환
rfind('찾을 문자열')	find() 함수와 반대로 '찾을 문자열'이 오른쪽 끝부터 시작하여 몇 번째에 있는지 반환
startswith('찾을 문자열')	'찾을 문자열'로 시작하는지 여부 반환
endswith('찾을 문자열')	'찾을 문자열'로 끝나는지 여부 반환

문자열 관련 주요 함수와 메서드



strip()	좌우 공백 삭제
rstrip()	오른쪽 공백 삭제
lstrip()	왼쪽 공백 삭제
split()	문자열을 공백이나 다른 문자로 나누어 리스트로 반환
isdigit()	문자열이 숫자인지 여부 반환
islower()	문자열이 소문자인지 여부 반환
isupper()	문자열이 대문자인지 여부 반환



문자열 메소드 (String Methods in Python)

string calculation

len()

min()

max()

count()

encoding/decoding

encode()

decode()

string search

startswith()

endswith()

find()

rfind()

index()

rindex()

number/ character

isalnum()

isalpha()

isdigit()

isnumeric()

isdecimal()

lower/ upper

islower()

isupper()

lower()

upper()

swapcase()

istitle()

title()

capitalize()

space/ strip

lstrip()

rstrip()

strip()

isspace()

center()

split/ join/ fill

split()

splitlines()

replace()

join()

zfill()

ljust()

rjust()

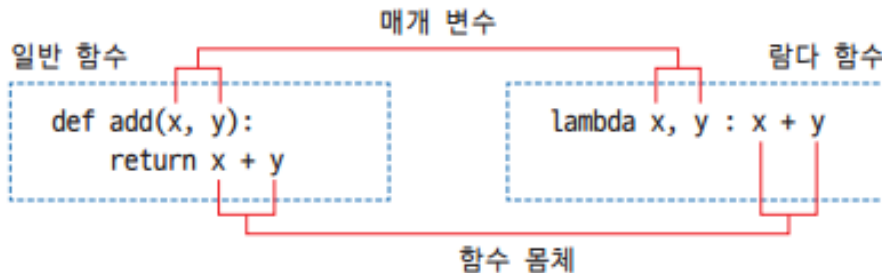
10.1 람다 함수

❖ 람다 함수 **lambda function**

- 1회용의 간단한 함수를 만드는 것

❖ 람다 표현식 **lambda expression**이라고도 불리우는 이름이 없는 함수

❖ 익명 함수 **anonymous function**



[그림 10-1] 일반 함수와 람다 함수: 일반 함수는 def 키워드와 함수명, 매개변수, 콜론, 그리고 몸체로 구성되어 있으나 람다 함수는 매개변수와 함수의 기능만을 가진다.

```
lambda [매개변수1, 매개변수2, ...] : [표현식]
```



```
(lambda [매개변수1, 매개변수2, ...] : [표현식])(인자1, 인자2, ...)
```

[그림 10-2] 람다 함수의 정의 방법과 인자 전달 방법

```
lambda [매개변수들] : {표현식}
```

- 람다 함수의 문법은 매우 간결하고 단순하다



코드 10-1 : add() 함수를 이용한 정수의 덧셈과 결과 반환

function_add.py

add() 함수를 이용한 덧셈

def add(x, y):

 return x + y

print('100과 200의 합 :', add(100, 200))

실행결과

100과 200의 합 : 300



코드 10-2 : 람다 함수를 정의하고 add라는 변수를 통해 참조하기

lambda_add.py

람다 함수를 이용한 덧셈

add = lambda x, y: x + y

print('100과 200의 합:', add(100, 200))

실행결과

100과 200의 합: 300

코드 10-3 : 인라인 람다 함수와 인자를 이용한 덧셈

inline_lambda_add.py

print('100과 200의 합:', (lambda x, y: x + y)(100, 200))

위와 같이 별도의 라인으로 람다 함수를 정의하지 않고
호출문 내에 람다 함수의 기능을 넣을 수 있다
이를 인라인 람다 함수라 한다

10.2 필터 함수 filter function

코드 10-4 : 19 이상의 값을 반환하는 `adult_func()` 함수와 필터 함수의 사용

`age_filter.py`

`# 19세 이상의 값이 들어오면 True, 그렇지 않으면 False를 반환`

```
def adult_func(n):
```

```
    if n >= 19:
```

```
        return True
```

```
    else:
```

```
        return False
```

```
ages = [34, 39, 20, 18, 13, 54]
```

```
print('성년 리스트 :')
```

```
for a in filter(adult_func, ages): # filter() 함수를 사용한 ages의 필터링
```

```
    print(a, end = ' ')
```

실행결과

성년 리스트 :

34 39 20 54

1. 알고리즘이 비교적 단순하다.
2. 별도의 내부 변수가 필요 없다.
3. 성년 리스트를 필터링한 후 더 이상 재사용할 필요가 없다.



코드 10-5 : 람다 함수를 이용한 간략화된 필터

age_lambda_filter.py


```
ages = [34, 39, 20, 18, 13, 54]
```

```
print('성년 리스트 :')
```

```
for a in filter(lambda x: x >= 19, ages): # filter() 함수를 사용한 ages의 필터
```

```
    print(a, end = ' ')
```

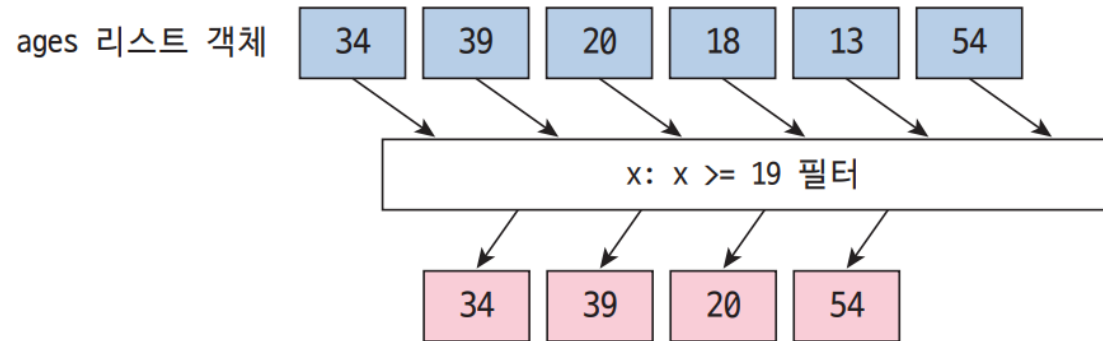
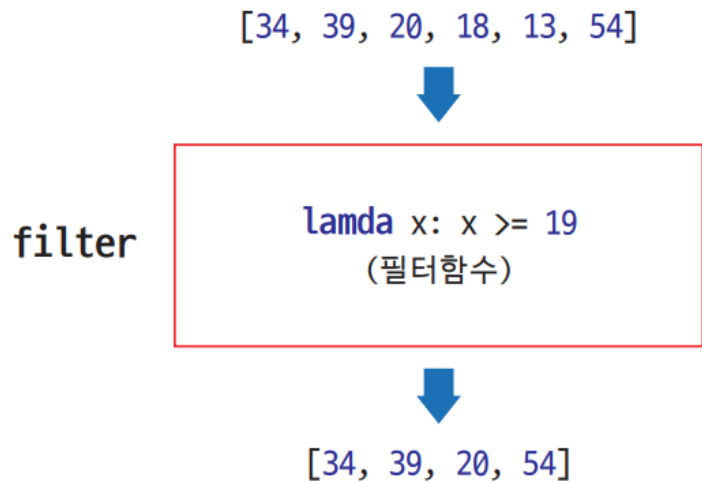
코드 10-4 보다 간결한 표현방법



실행결과

성년 리스트 :

34 39 20 54



[그림 10-4] 반복 가능한 객체와 필터 함수에 의해 변환된 반복자 객체

[그림 10-3] filter() 함수의 동작과 람다 함수의 역할



코드 10-7 : 람다 함수를 이용한 음수 값 추출기능 1

minus_filter_func.py

```
def minus_func(n): # n이 음수이면 True, 그렇지 않으면 False를 반환
    if n < 0:
        return True
    else:
        return False

n_list = [-30, 45, -5, -90, 20, 53, 77, -36]
minus_list = [] # 음수를 저장할 리스트
for n in filter(minus_func, n_list):
    minus_list.append(n)
print('음수 리스트 :', minus_list)
```



코드 10-8 : 람다 함수를 이용한 음수 값 추출기능 2

lambda_filter_minus1.py

```
n_list = [-30, 45, -5, -90, 20, 53, 77, -36]
minus_list = []

for n in filter(lambda x: x < 0, n_list):
    minus_list.append(n)

print('음수 리스트 :', minus_list)
```

실행결과

음수 리스트 : [-30, -5, -90, -36]

코드 10-9 : 람다 함수를 이용한 음수 값 추출기능 3

lambda_filter_minus2.py

```
n_list = [-30, 45, -5, -90, 20, 53, 77, -36]
minus_list = list(filter(lambda x: x < 0, n_list))
print('음수 리스트 :', minus_list)
```

10.3 맵 함수 `map function`

코드 10-10 : 맵 함수를 이용한 제곱값 리스트 생성

```
square_map_for.py
```

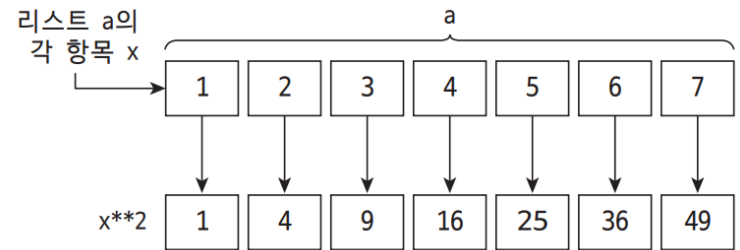
```
a = [1, 2, 3, 4, 5, 6, 7]
```

```
square_a = []
```

```
for n in a:
```

```
    square_a.append(n**2) # n의 제곱을 square_a 리스트에 추가
```

```
print(square_a)
```



[그림 10-5] 원본 리스트 `a`와 각 항목 `x`에 대한 제곱 연산이 적용된 리스트

실행결과

```
[1, 4, 9, 16, 25, 36, 49]
```

- [코드 10-10]은 map() 함수와 square() 함수를 사용하면 다음 [코드 10-11]과 같이 간단하게 수정할 수 있다.

map(적용시킬_함수, 반복가능 객체, ...)

코드 10-11 : 맵 함수와 square() 호출을 이용한 제곱값 리스트 생성

square_map_func.py

```
def square(x):  
    return x ** 2
```

```
a = [1, 2, 3, 4, 5, 6, 7]
```

a의 각 항목에 대해 square 함수의 반환값을 적용

```
square_a = list(map(square, a))
```

```
print(square_a)
```

실행결과

[1, 4, 9, 16, 25, 36, 49]



코드 10-12 : 맵 함수와 람다 함수를 이용한 제곱값 리스트 생성

square_map_lambda.py

```
a = [1, 2, 3, 4, 5, 6, 7]
square_a = list(map(lambda x: x**2, a))
print(square_a)
```

실행결과

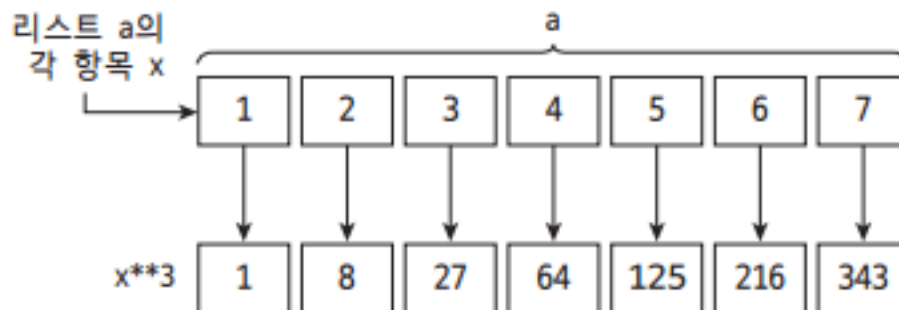
[1, 4, 9, 16, 25, 36, 49]



```
a = [1, 2, 3, 4, 5, 6, 7]
```

```
square_a = list(map(lambda x: x**2, a))
```

```
cubic_a = list(map(lambda x: x**3, a))
```



[그림 10-6] 람다 함수를 사용한 매핑 함수의 동작

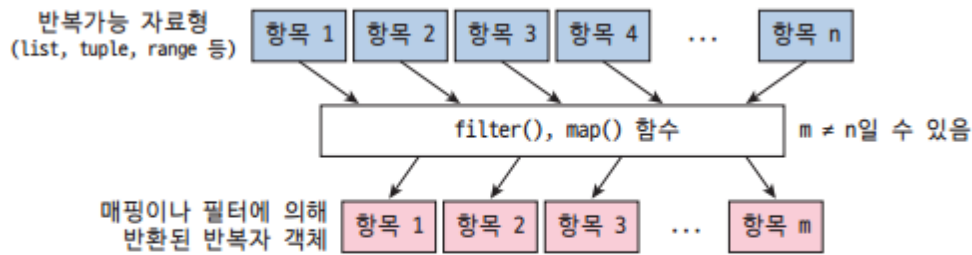
- `map()` 함수와 람다 함수를 사용하면 간결한 표현식으로 리스트의 요소들을 변환할 수 있다.



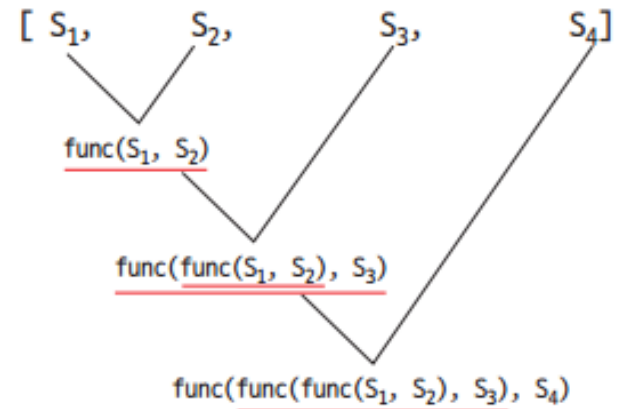
NOTE : 람다 함수의 장점과 단점

- **장점:** 람다 함수는 축약된 표현이 가능하기 때문에 코드가 간결해 진다. 일반 함수의 경우 재사용을 위하여 함수를 위한 이름과 이 메모리 공간의 할당이 필요하지만 람다 함수는 메모리 공간을 별도로 할당하지 않기 때문에 메모리를 절약할 수 있다. 즉 람다 함수는 익명 함수이므로 표현식 내에서 사용된 후 다음 코드로 이동하면 힙 메모리에서 사라지게 된다.
- **단점:** lambda 선언문 뒤에 쉼표와 콜론 수식으로 구성되어 가독성이 떨어진다. 즉 람다식은 그 자체에 쉼표(,)를 포함하고 있기 때문에 다른 매개변수와 시각적인 구분이 모호해지고, 읽고 이해하기가 어렵다.

10.4 리듀스reduce 함수



[그림 10-7] 반복가능한 자료형과 매핑 함수에 의해 변환된 반복자 객체



[그림 10-8] reduce() 함수의 동작



코드 10-13 : reduce() 함수를 사용한 멤버 덧셈

reduce_lambda_sum.py

```
from functools import reduce
```

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

```
n = reduce(lambda x, y: x + y, a)
```

```
print(n)
```

실행결과

10

$((1 + 2) + 3) + 4 \Rightarrow 10$



코드 10-14 : 람다 함수와 리듀스를 이용한 멤버값의 곱셈

reduce_lambda_mult.py

```
from functools import reduce

a = [1, 2, 3, 4]
n = reduce(lambda x, y: x * y, a)
print(n)
```

실행결과

24

$((1 * 2) * 3) * 4 \Rightarrow 24$

2. 람다 함수

❖ filter 함수

- 리스트 요소 중 조건에 맞는 것만을 골라냄
- 첫 번째 인수 : 조건 지정하는 함수
- 두 번째 인수 : 대상 리스트

filter

```
def flunk(s):  
    return s < 60  
  
score = [ 45, 89, 72, 53, 94 ]  
for s in filter(flunk, score):  
    print(s)
```

실행결과

45
53

- 직접 정의한 flunk 함수
 - 점수 s를 인수로 받아 60 미만인지 조사

2. 람다 함수

❖ map 함수

- 모든 요소에 대한 변환함수 호출, 새 요소 값으로 구성된 리스트 생성
- 첫 번째 인수 : 조건 지정하는 함수
- 두 번째 인수 : 대상 리스트

map

```
def half(s):  
    return s / 2  
  
score = [ 45, 89, 72, 53, 94 ]  
for s in map(half, score):  
    print(s, end = ', ')
```

실행결과

22.5, 44.5, 36.0, 26.5, 47.0,

- 직접 정의한 half 함수
 - 인수로 전달받은 s를 절반으로 나누어 리턴

2. 람다 함수

❖ 람다 함수

- 이름 없고 입력과 출력만으로 함수를 정의하는 축약된 방법

■ `lambda` 인수 : 식

- 인수는 여러 개 가질 수 있음

호출 시 전달할 리턴 값

호출 시 전달받을
아규먼트(생략 가능)

```
lambda x:x + 1
```

lambda

```
score = [ 45, 89, 72, 53, 94 ]  
for s in filter(lambda x:x < 60, score):  
    print(s)
```




10.5 리스트와 축약 표현

❖ 리스트 축약표현 `list comprehension`의 문법은 다음과 같다.

```
[ {표현식} for {변수} in {반복자/연속열} if {조건 표현식} ]
```

- 리스트 축약문법에서 다음과 같이 if 조건 표현식은 생략 가능하다

```
[ {표현식} for {변수} in {반복자/연속열} ]
```



코드 10-15 : 맵과 람다 함수를 이용한 리스트의 제곱 구하기

list_map_lambda_square.py

```
a = [1, 2, 3, 4, 5, 6, 7]    # 연속된 값을 가지는 리스트
a = list(map(lambda x: x**2, a)) # 리스트의 각 요소에 대하여 람다 함수 적용
print(a)
```

코드 10-16 : 리스트 축약 표현식을 이용한 리스트의 제곱 구하기

list_comp_square.py

```
a = [1, 2, 3, 4, 5, 6, 7]    # 연속된 값을 가지는 리스트
a = [x**2 for x in a]        # 리스트의 각 요소에 대하여 x**2를 적용
print(a)
```

코드 10-17 : 리스트 축약 표현식과 range를 이용한 리스트의 제곱 구하기

list_comp_range_square.py

```
a = [x**2 for x in range(1, 8)]
print(a)
```

- 세 코드의 실행 결과는 다음과 같이 동일하다.

실행결과

[1, 4, 9, 16, 25, 36, 49]



코드 10-18 : 문자열 각각을 대문자로 바꾸는 기능

upper_chars_from_string.py

```
st = 'Hello World'
```

```
s_list = [x.upper() for x in st] # 문자열 각각에 대해 upper() 메소드 적용
```

```
print(s_list)
```

실행결과

```
['H', 'E', 'L', 'L', 'O', ' ', 'W', 'O', 'R', 'L', 'D']
```

10.5.2 if 조건식을 이용한 필터링

코드 10-19 : list() 함수, filter() 함수, 람다 함수를 이용한 필터링

age_lambda_filter_list.py

```
ages = [34, 39, 20, 18, 13, 54]
adult_ages = list(filter(lambda x: x >= 19, ages))
print('성년 리스트 : ', adult_ages)
```

실행결과

성년 리스트 : [34, 39, 20, 54]

- 위의 코드는 다음 [코드10-20]과 같은 리스트 축약 표현을 사용할 수 있다.(결과는 위와 동일함)

코드 10-20 : 리스트 축약표현을 이용한 필터링

age_list_comp_filter.py

```
ages = [34, 39, 20, 18, 13, 54]
print('성년 리스트 : ', [x for x in ages if x >= 19])
```



코드 10-21 : list() 함수, filter() 함수, 람다 함수를 이용한 필터링

lambda_filter_minus2.py

```
n_list = [-30, 45, -5, -90, 20, 53, 77, -36]
minus_list = list(filter(lambda x: x < 0, n_list))
print('음수 리스트 :', minus_list)
```

실행결과

음수 리스트 : [-30, -5, -90, -36]

- 위의 코드는 다음 [코드10-20]과 같은 리스트 축약 표현을 사용하여 동일한 결과를 얻을 수 있다.

코드 10-22 : 리스트 축약 표현과 if 조건식을 이용한 필터링

list_comp_if_condition.py

```
n_list = [-30, 45, -5, -90, 20, 53, 77, -36]
minus_list = [x for x in n_list if x < 0]
print('음수 리스트 :', minus_list)
```



❖ 필터링 된 값에 대해 $x * x$ 를 통해 각 값의 제곱을 구한 후 이 값을 리스트에 넣는 과정이다

대화창 실습 : 리스트의 축약 표현 실습

```
>>> [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]
```

```
>>> [x for x in range(10) if x % 2 == 0] # 0에서 9까지 숫자 중 짝수 값
```

```
[0, 2, 4, 6, 8]
```

```
>>> [x for x in range(10) if x % 2 == 1] # 0에서 9까지 숫자 중 홀수 값
```

```
[1, 3, 5, 7, 9]
```

```
>>> [x * x for x in range(10) if x % 2 == 0] # 0에서 9까지 숫자 중 짝수의 제곱 값
```

```
[0, 4, 16, 36, 64]
```

```
>>> [x * x for x in range(10) if x % 2 == 1] # 0에서 9까지 숫자 중 홀수의 제곱 값
```

```
[1, 9, 25, 49, 81]
```



- ❖ 여러 개의 정수 문자열을 입력받아 이를 int() 함수를 사용해서 정수형 값을 가진 리스트로 변환하는 것도 손쉽게 할 수 있다.

대화창 실습 : 여러 개의 정수를 리스트로 입력받는 방법

```
>>> s = input('정수 5개를 입력하세요 : ').split()
```

```
정수 5개를 입력하세요 : 10 20 30 40 50
```

```
>>> lst = [int(x) for x in s]
```

```
>>> lst
```

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

```
>>> [int(x) for x in input('정수를 여러개 입력하세요 : ').split()]
```

```
정수를 여러개 입력하세요 : 1 2 3
```

```
[1, 2, 3]
```



코드 10-23 : 두 리스트를 곱하여 새 리스트를 생성하는 코드

product_xy_for_loop.py

```
product_xy = []  
for x in [1, 2, 3]: # 이중 for 루프를 통해 두 리스트 원소의 곱을 모두 구함  
    for y in [2, 4, 6]:  
        product_xy.append(x * y)  
print(product_xy)
```

리스트 축약 표현을 이용하여
한 줄의 코딩으로 표현 가능

실행결과

[2, 4, 6, 4, 8, 12, 6, 12, 18]

코드 10-24 : 리스트 축약을 이용한 두 리스트의 곱하기 기능

product_xy_comprehension.py

```
product_xy = [x * y for x in [1, 2, 3] for y in [2, 4, 6]]  
print(product_xy)
```




대화창 실습 : 리스트의 축약 표현을 사용한 2와 3의 배수 구하기

```
>>> [n for n in range(1, 31) if n % 2 == 0 if n % 3 == 0]
```

```
[6, 12, 18, 24, 30]
```

필요에 따라서 더 많은 if를 추가하여 조건을 더 줄 수
있다(1에서 30까지의 수 중에서 2, 3, 5의 배수 구하기)

대화창 실습 : 리스트의 축약 표현을 사용한 2와 3과 5의 배수 구하기

```
>>> [n for n in range(1, 31) if n % 2 == 0 if n % 3 == 0 if n % 5 == 0]
```

```
[30]
```



10.8 반복가능 객체를 위한 내장함수

- ❖ 파이썬의 반복가능 `iterable` 객체는 다양한 내장함수들을 적용할 수 있다.
- ❖ 앞서 살펴봤던 `min()`이나 `max()`와 같은 함수는 반복가능 객체를 인자로 받아서 최소값과 최대값을 반환하는데 이들 외에도 `all()`, `any()`, `ascii()`, `bool()`, `filter()`, `iter()`와 같은 고급 내장함수도 제공되고 있다

❖ `all()` 함수는 반복 가능한 항목들이 모두 참일 때 참을 반환한다.

대화창 실습 : 리스트에 대한 `all()` 함수 적용 결과

```
>>> l1 = [1,2,3,4] # 모든 요소가 0이 아님
```

```
>>> l2 = [0,2,4,8] # 한 요소만 0임
```

```
>>> l3 = [0,0,0,0] # 모든 요소가 0임
```

```
>>> all(l1) # 모든 요소가 true(0이 아닌 값)일 때만 True를 반환함
```

```
True
```

```
>>> all(l2)
```

```
False
```

```
>>> all(l3)
```

```
False
```



❖ `any()`는 임의의 반복 가능한 항목들 중에서 참이 하나라도 있을 경우 참을 반환한다

대화창 실습 : 리스트에 대한 `any()` 함수 적용 결과

```
>>> any(l1)      # 항목들 중 하나라도 true(0이 아닌 값)일 경우 True를 반환함
```

```
True
```

```
>>> any(l2)      # 항목들 중 하나라도 true(0이 아닌 값)일 경우 True를 반환함
```

```
True
```

```
>>> any(l3)      # 항목들이 모두 false(0의 값)이므로 False를 반환함
```

```
False
```



- ❖ `bool()`은 값(리스트)을 부울 값으로 변환한다. 즉 리스트의 항목 유무를 `True`와 `False`로 알려준다

대화창 실습 : 리스트에 대한 `any()` 함수 적용 결과

```
>>> bool(l1)      # 리스트에 원소가 있으면 True를 반환함
```

```
True
```

```
>>> bool(l2)      # 리스트에 원소가 있으면 True를 반환함
```

```
True
```

```
>>> bool(l3)
```

```
True
```

```
>>> l4 = []        # 비어있는 리스트 l4
```

```
>>> bool(l4)      # 리스트에 원소가 없으면 False를 반환함
```

```
False
```



대화창 실습 : 문자열을 리스트 형으로 변환시킴

```
>>> char_list = list('hello')
```

```
>>> char_list
```

```
['h', 'e', 'l', 'l', 'o']
```

대화창 실습 : 공백 단위의 리스트를 만드는 split() 메소드

```
>>> words = 'Python은 아름다운 언어입니다.'
```

```
>>> words_list = words.split()
```

```
>>> words_list
```

```
['Python은', '아름다운', '언어입니다.']
```



대화창 실습 : 마침표로 구분된 리스트를 만드는 split() 메소드

```
>>> time_str = '2019.02.20'
```

```
>>> time_str.split('.')
```

```
['2019', '02', '20']
```

대화창 실습 : join() 메소드를 이용한 리스트의 연결

```
>>> time_list = ['2019', '02', '21']
```

```
>>> time_list
```

```
['2019', '02', '21']
```

```
>>> ''.join(time_list)
```

```
'2019.02.21'
```

```
>>> ','.join(time_list)
```

```
'2019,02,21'
```



■ 지능형 리스트 다루기

- 지능형 리스트(list comprehension)의 기본 개념은 기존 리스트형을 사용하여 간단하게 새로운 리스트를 만드는 기법이다. 리스트와 for문을 한 줄에 사용할 수 있는 장점이 있다.

일반적인 반복문 + 리스트

```
>>> result = []
>>> for i in range(10):
...     result.append(i)
...
>>> result
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

리스트 컴프리헨션

```
>>> result = [i for i in range(10)]
>>> result
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```




■ 지능형 리스트용법 : 조건체크

- 필터링은 if문과 함께 사용하는 리스트 컴프리헨션이다. 일반적으로 짝수만 저장하기 위해서는 다음과 같은 코드를 작성해야 한다.

일반적인 반복문 + 리스트

```
>>> result = []
>>> for i in range(10):
...     if i % 2 == 0:
...         result.append(i)
...
>>> result
[0, 2, 4, 6, 8]
```

리스트 컴프리헨션

```
>>> result = [i for i in range(10) if i % 2 == 0]
>>> result
[0, 2, 4, 6, 8]
```



■ 지능형 리스트용법 : 조건체크

- 다음 코드를 보면, 기존 리스트 컴프리헨션문 끝에 `if i % 2 == 0` 을 삽입하여 해당 조건을 만족할 때만 `i`를 추가할 수 있게 한다. `else`문과 함께 사용하면 해당 조건을 만족하지 않을 때는 다른 값을 할당할 수 있다.

```
>>> result = [i if i % 2 == 0 else 10 for i in range(10)]  
>>> result  
[0, 10, 2, 10, 4, 10, 6, 10, 8, 10]
```

- **if 문을 앞으로 옮겨 else절과 함께 사용하면, 조건을 만족할 때는 if 앞의 i 변수의 값을 그리고 만족하지 않을 때는 else 뒤에 값을 할당하는 코드를 작성할 수 있다.**



```
>>> [ i for i in range(10) ]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> [ i+10 for i in range(10) ]
[10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
>>> [ i*i for i in range(10) ]
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
>>> [ i for i in range(10) if i % 2 ]
[1, 3, 5, 7, 9]
>>> [ i*i for i in range(10) if not i % 2 ]
[0, 4, 16, 36, 64]
>>> [ 'A'+str(i) for i in range(10) if i % 2 ]
['A1', 'A3', 'A5', 'A7', 'A9']
>>> [ 'A'+str(i) if i % 2 else 'B'+str(i) for i in range(10) ]
['B0', 'A1', 'B2', 'A3', 'B4', 'A5', 'B6', 'A7', 'B8', 'A9']
>>> [ 'big' if i > 5 else 'small' for i in range(1,10) ]
['small', 'small', 'small', 'small', 'small', 'big', 'big', 'big', 'big']
```



[값표현식 for 요소 in 반복자]
[값표현식 for 요소 in 반복자 if 조건식]
[값표현식1 if 조건식 else 값표현식2 for 요소 in 반복자]

{ 값표현식 for 요소 in 반복자 }
{ 값표현식 for 요소 in 반복자 if 조건식 }
{ 값표현식1 if 조건식 else 값표현식2 for 요소 in 반복자 }

{ 키표현식:값표현식 for 요소 in 반복자 }
{ 키표현식:값표현식 for 요소 in 반복자 if 조건식 }
{ 키표현식:값표현식1 if 조건식 else 값표현식2 for 요소 in 반복자 }



■ 지능형 리스트용법 : 중첩 반복문

- 리스트 컴프리헨션에서도 기존처럼 리스트 2개를 섞어 사용할 수 있다.
- 다음 코드와 같이 2개의 for문을 만들 수 있다.

```
>>> word_1 = "Hello"
>>> word_2 = "World"
>>> result = [i + j for i in word_1 for j in word_2]           # 중첩 반복문
>>> result
['HW', 'Ho', 'Hr', 'Hl', 'Hd', 'eW', 'eo', 'er', 'el', 'ed', 'lW', 'lo', 'lr', 'll', 'ld', 'lW',
'lo', 'lr', 'll', 'ld', 'oW', 'oo', 'or', 'ol', 'od']
```

- ➡ 위 코드를 보면, word_1에서 나오는 값을 먼저 고정한 후, word_2의 값을 하나씩 가져와 결과를 생성한다.



■ 지능형 리스트용법: 중첩 반복문

- 중첩 반복문에서도 필터링을 적용할 수 있다. 다음과 같이 반복문 끝에 f문을 추가하면 된다.

```
>>> case_1 = ["A", "B", "C"]
>>> case_2 = ["D", "E", "A"]
>>> result = [i + j for i in case_1 for j in case_2 if not(i==j)]
>>> result
['AD', 'AE', 'BD', 'BE', 'BA', 'CD', 'CE', 'CA']
```



■ 지능형 리스트용법 : 이차원 리스트

- 비슷한 방식으로 이차원 리스트(two-dimensional list) 를 만들 수 있다. 앞 중첩 반복문의 예시 코드 결과는 일차원 리스트(one-dimensional list) 였다. 그렇다면 하나의 정보를 열row 단위로 저장하는 이차원 리스트는 어떻게 만들 수 있을까? 먼저 다음 코드를 보자.

```
>>> words = 'The quick brown fox jumps over the lazy dog'.split()
>>> print(words)
['The', 'quick', 'brown', 'fox', 'jumps', 'over', 'the', 'lazy', 'dog']
>>> stuff = [[w.upper(), w.lower(), len(w)] for w in words]      # 리스트의 각 요소를 대문
자, 소문자, 길이로 변환하여 이차원 리스트로 변환
```

- ➡ 가장 간단한 방법은 위 코드처럼 대괄호 2개를 사용하는 것이다. 이 코드는 기존 문장을split() 함수로 분리하여 리스트로 변환한 후, 각 단어의 대문자, 소문자, 길이를 하나의 리스트로 따로 저장하는 방식이다.



■ 지능형 리스트용법 : 이차원 리스트

- 저장한 후 결과를 출력하면, 다음과 같이 이차원 리스트를 생성할 수 있다.

```
>>> for i in stuff:  
...     print (i)  
...  
['THE', 'the', 3]  
['QUICK', 'quick', 5]  
['BROWN', 'brown', 5]  
['FOX', 'fox', 3]  
['JUMPS', 'jumps', 5]  
['OVER', 'over', 4]  
['THE', 'the', 3]  
['LAZY', 'lazy', 4]  
['DOG', 'dog', 3]
```




■ 지능형 리스트용법 : 이차원 리스트

- 다른 방법으로 for문 2개를 붙여 사용할 수도 있다. 여기서 한 가지 주의할 점은 for문 2개를 붙여 사용하면 대괄호의 위치에 따라 for문의 실행이 달라진다는 것이다. 이전에 배웠던 for문은 앞에 있는 for문이 먼저 실행된 후, 뒤의 for문이 실행되었다. 그래서 다음 코드의 경우 A가 먼저 고정되고, D, E, A를 차례대로 붙여 결과가 출력된다.

```
>>> case_1 = ["A", "B", "C"]
>>> case_2 = ["D", "E", "A"]
>>> result = [i + j for i in case_1 for j in case_2]
>>> result
['AD', 'AE', 'AA', 'BD', 'BE', 'BA', 'CD', 'CE', 'CA']
```



■ 지능형 리스트용법 : 이차원 리스트

- 이차원 리스트를 만들기 위해서는 대괄호를 하나 더 사용해야 한다. 그리고 그와 동시에 먼저 작동하는 for문의 순서가 달라진다. 다음 코드는 위의 코드와 달리 리스트 안에 `[i + j for i in case_1]`이 하나 더 존재한다. 따라서 먼저 나온 for문이 고정되는 것이 아니라, 뒤의 for문이 고정된다. 즉, A부터 고정되는 것이 아니라 case_2의 첫 번째 요소인 D가 고정되고 A, B, C가 차례로 D 앞에 붙는다. 결과를 보면 이차원 리스트 형태로 출력된 것을 확인할 수 있다.

```
>>> result = [[ i + j for i in case_1] for j in case_2]
>>> result
[['AD', 'BD', 'CD'], ['AE', 'BE', 'CE'], ['AA', 'BA', 'CA']]
```



■ 지능형 리스트용법 : 이차원 리스트

- 다음 두 코드는 꼭 구분해야 한다.

①	②
① [i + j for i in case_1 for j in case_2]	
② [[i + j for i in case_1] for j in case_2]	
②	①

- ➔ 첫 번째 코드는 일차원 리스트를 만드는 코드로, 앞의 for문이 먼저 실행된다. 두 번째 코드는 이차원 리스트를 만드는 코드로, 뒤의 for문이 먼저 실행된다. 이 두 코드의 차이를 꼭 이해하고 넘어가자.

1. 사전

❖ 사전 컴프리헨션 (Dictionary Comprehension) – 지능형 사전

{ 키와 값에 대한 수식 for 변수 in 대상 if 조건 }

```
score_dict = {t[0]: t[1] for t in score_tuples}
print(score_dict)
```

```
score_dict = {k : v for k, v in score_tuples}
print(score_dict)
```

```
score_dict = {k : v for k, v in score_tuples if len(k) > 5}
print(score_dict)
```



■ 지능형 리스트의 성능

- 두 코드의 성능을 비교하기 위해 리눅스 계열에서 사용하는 time 명령어의 결과를 캡처한 것이다. 코드의 총 실행 시간을 확인할 수 있다.

```
real    0m10.230s
user    0m10.203s
sys     0m0.023s
```

(a) 일반적인 반복문 + 리스트

```
real    0m7.788s
user    0m7.762s
sys     0m0.021s
```

(b) 리스트 컴프리헨션

[코드의 실행 시간을 통한 성능 비교]

❖ 집합 컴프리헨션 (Set Comprehension) – 지능형 집합

{값에 대한 수식 for 변수 in 대상 if 조건}

```
a = {i for i in range(1, 101) if i % 3 == 0}
```

```
b = {i for i in range(1, 101) if i % 5 == 0}
```

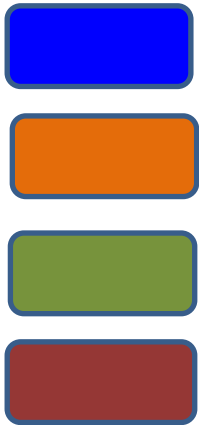
```
listv = [dan * num for dan in range(1, 10) for num in range(1, 10)]
```

```
setv = { dan * num for dan in range(1, 10) for num in range(1, 10)}
```

객체 정리



함수



메서드

