



파이썬 프로그래밍 [3]

[if조건문 / 리스트 / 반복문 / 딕셔너리]



SW융합학부

강희국

※ 수강생을 위한 참고자료로 제3자에 대한 배포를 금지합니다. 법적인 문제 발생 시 배포자에게 책임이 있습니다.

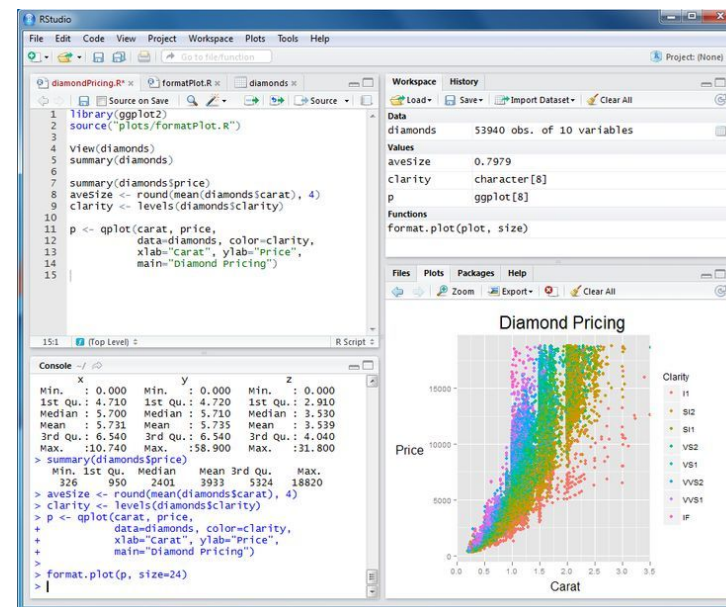
1 if조건문과 불자료형

2 if~else와 elif구문

3 리스트와 반복문

4 딕셔너리와 반복문

5 반복문과 while 반복문



1. 불 자료형과 if 조건문



[참고] 윤인성, "혼자 공부하는 파이썬", 한빛미디어

[핵심 키워드] 불, 비교 연산자, 논리 연산자, if 조건문

[핵심 포인트] 프로그래밍 언어에는 기본적인 자료형으로 참과 거짓을 나타내는 값이 있으며, 이를 불(boolean)이라 한다. 불 자료를 만드는 방법과 이에 관련된 연산자에 대해 알아본다.

● Boolean

- 불린 / 불리언 / 불
- True와 False 값만 가질 수 있음

```
>>> print(True)
True
>>> print(False)
False
```

- 비교 연산자를 통해 만들 수 있음

연산자	설명
==	같다
!=	다르다
<	작다

연산자	설명
>	크다
<=	작거나 같다
>=	크거나 같다

➤ 숫자 또는 문자열에 적용

```
>>> print(10 == 100)
False
>>> print(10 != 100)
True
>>> print(10 < 100)
True
>>> print(10 > 100)
False
>>> print(10 <= 100)
True
>>> print(10 >= 100)
False
```

조건식	의미	결과
<code>10 == 100</code>	10과 100은 같다	거짓
<code>10 != 100</code>	10과 100은 다르다	참
<code>10 < 100</code>	10은 100보다 작다	참
<code>10 > 100</code>	10은 100보다 크다	거짓
<code>10 <= 100</code>	10은 100보다 작거나 같다	참
<code>10 >= 100</code>	10은 100보다 크거나 같다	거짓

- 문자열에도 비교 연산자 적용 가능

```
>>> print("가방" == "가방")
True
>>> print("가방" != "하마")
True
>>> print("가방" < "하마")
True
>>> print("가방" > "하마")
False
```

➤ 불끼리 논리 연산자 사용 가능

연산자	의미	설명
not	아니다	불을 반대로 전환합니다.
and	그리고	피연산자 두 개가 모두 참일 때 True를 출력하며, 그 외는 모두 False를 출력합니다.
or	또는	피연산자 두 개 중에 하나만 참이라도 True를 출력하며, 두 개가 모두 거짓일 때만 False를 출력합니다.

● not 연산자

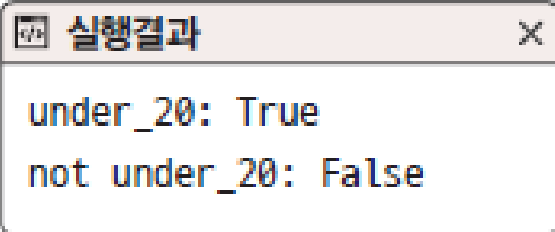
➤ 단항 연산자

➤ 참과 거짓 반대로 바꿈

```
>>> print(not True)
False
>>> print(not False)
True
```


➤ 예시 - not 연산자 조합하기

```
01 x = 10
02 under_20 = x < 20
03 print("under_20:", under_20)
04 print("not under_20:", not under_20)
```



실행결과

under_20: True
not under_20: False

● and 연산자와 or 연산자

➤ and 연산자는 양쪽 변의 값이 모두 참일 때만 True를 결과로 냄

➤ and 연산자

좌변	우변	결과
True	True	True
True	False	False
False	True	False
False	False	False

➤ or 연산자

좌변	우변	결과
True	True	True
True	False	True
False	True	True
False	False	False

➤ 예시 - and 연산자와 or 연산자

"사과 그리고 배 가져와!"

"사과 또는 배 가져와!"

"치킨(True) 그리고 쓰레기(False) 가져와!"

"치킨(True) 또는 쓰레기(False) 가져와!"

```
>>> print(True and True)
True
>>> print(True and False)
False
>>> print(False and True)
False
>>> print(False and False)
False
>>> print(True or True)
True
>>> print(True or False)
True
>>> print(False or True)
True
>>> print(False or False)
False
```

● and 연산자



● or 연산자



● if 조건문

- 조건에 따라 코드 실행하거나 실행하지 않게 할 때 사용하는 구문
- 조건 분기

if 불 값이 나오는 표현식: → if의 조건문 뒤에는 반드시 콜론(:)을 붙여줘야 합니다.

□□□□ 불 값이 참일 때 실행할 문장

□□□□ 불 값이 참일 때 실행할 문장

□□□□는 들여쓰기 4칸

↓
if문 다음 문장은 4칸 들여쓰기 후 입력합니다.

➤ 예시

```
>>> if True: Enter
    print("True입니다...!") Enter
    print("정말 True입니다...!") Enter
Enter
True입니다...!
정말 True입니다...!
```

```
>>> if False: Enter
    print("False입니다...!") Enter
Enter
>>>
```

➤ 예시 - 조건문의 기본 사용

```
01  # 입력을 받습니다.  
02  number = input("정수 입력> ")  
03  number = int(number)  
04  
05  # 양수 조건  
06  if number > 0:  
07      print("양수입니다")  
08  
09  # 음수 조건  
10  if number < 0:  
11      print("음수입니다")  
12  
13  # 0 조건  
14  if number == 0:  
15      print("0입니다")
```

실행결과 1

정수 입력> 273 Enter

양수입니다

실행결과 2

정수 입력> -52 Enter

음수입니다

실행결과 3

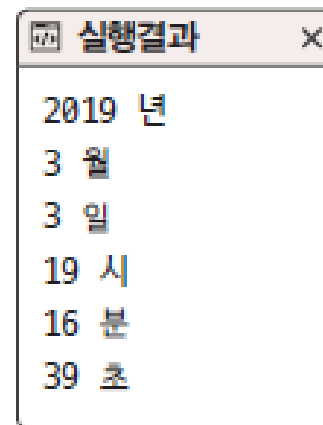
정수 입력> 0 Enter

0입니다

➤ 예시 - 날짜/시간 출력하기

- datetime.datetime.now() 함수

```
01  # 날짜/시간과 관련된 기능을 가져옵니다.  
02  import datetime  
03  
04  # 현재 날짜/시간을 구합니다.  
05  now = datetime.datetime.now()  
06  
07  # 출력합니다.  
08  print(now.year, "년")  
09  print(now.month, "월")  
10  print(now.day, "일")  
11  print(now.hour, "시")  
12  print(now.minute, "분")  
13  print(now.second, "초")
```



실행결과

2019 년
3 월
3 일
19 시
16 분
39 초

➤ 예시 - 날짜/시간을 한 줄로 출력하기

```
01  # 날짜/시간과 관련된 기능을 가져옵니다.  
02  import datetime  
03  
04  # 현재 날짜/시간을 구합니다.  
05  now = datetime.datetime.now()  
06  
07  # 출력합니다.  
08  print("{}년 {}월 {}일 {}시 {}분 {}초".format(  
09      now.year,  
10      now.month,  
11      now.day,  
12      now.hour,  
13      now.minute,  
14      now.second  
15  ))
```

실행결과

2019년 3월 3일 19시 18분 45초

➤ 예시 - 오전과 오후를 구분하는 프로그램

```
01  # 날짜/시간과 관련된 기능을 가져옵니다.  
02  import datetime  
03  
04  # 현재 날짜/시간을 구합니다.  
05  now = datetime.datetime.now()  
06  
07  # 오전 구분  
08  if now.hour < 12:  
09      print("현재 시각은 { }시로 오전입니다!".format(now.hour))  
10  
11  # 오후 구분  
12  if now.hour >= 12:  
13      print("현재 시각은 { }시로 오후입니다!".format(now.hour))
```

실행결과

현재 시각은 19시로 오후입니다!

➤ 예시 - 계절을 구분하는 프로그램

```
01  # 날짜/시간과 관련된 기능을 가져옵니다.  
02  import datetime  
03  
04  # 현재 날짜/시간을 구합니다.  
05  now = datetime.datetime.now()  
06  
07  # 봄 구분  
08  if 3 <= now.month <= 5:  
09      print("이번 달은 {}월로 봄입니다!".format(now.month))  
10  
11  # 여름 구분  
12  if 6 <= now.month <= 8:
```

```
13     print("이번 달은 {}월로 여름입니다!".format(now.month))
14
15     # 가을 구분
16     if 9 <= now.month <= 11:
17         print("이번 달은 {}월로 가을입니다!".format(now.month))
18
19     # 겨울 구분
20     if now.month == 12 or 1 <= now.month <= 2:
21         print("이번 달은 {}월로 겨울입니다!".format(now.month))
```

실행결과

이번 달은 3월로 봄입니다!

● if 조건문의 형식

if 불 값이 나오는 표현식:

□□□□ 불 값이 참일 때 실행할 문장

□□□□는 들여쓰기 4칸

➤ 예시 - 끝자리로 짝수와 홀수 구분

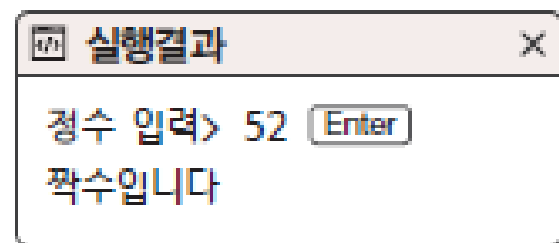
```
01  # 입력을 받습니다.  
02  number = input("정수 입력> ")  
03  
04  # 마지막 자리 숫자를 추출  
05  last_character = number[-1]  
06  
07  # 숫자로 변환하기  
08  last_number = int(last_character)  
09
```

```
10  # 짝수 확인
11  if last_number == 0 \
12      or last_number == 2 \
13      or last_number == 4 \
14      or last_number == 6 \
15      or last_number == 8:
16      print("짝수입니다")
17
18  # 홀수 확인
19  if last_number == 1 \
20      or last_number == 3 \
21      or last_number == 5 \
22      or last_number == 7 \
23      or last_number == 9:
24      print("홀수입니다")
```



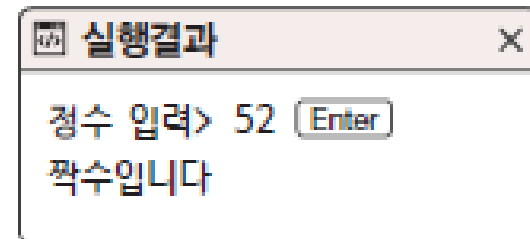
➤ 예시 - in 연산자를 활용한 수정

```
01  # 입력을 받습니다.  
02  number = input("정수 입력> ")  
03  last_character = number[-1]  
04  
05  # 짝수 조건  
06  if last_character in "02468":  
07      print("짝수입니다")  
08  
09  # 홀수 조건  
10  if last_character in "13579":  
11      print("홀수입니다")
```



➤ 예시 - 나머지 연산자를 활용한 짝수와 홀수 구분

```
01  # 입력을 받습니다.  
02  number = input("정수 입력> ")  
03  number = int(number)  
04  
05  # 짝수 조건  
06  if number % 2 == 0:  
07      print("짝수입니다")  
08  
09  # 홀수 조건  
10  if number % 2 == 1:  
11      print("홀수입니다")
```



- **불** : 파이썬의 기본 자료형으로 True와 False 나타내는 값
- **비교 연산자** : 숫자 또는 문자열에 적용하며 대소 비교하는 연산자
- **논리 연산자** : not, and, or 연산자 있으며, 불 만들 때 사용
- **if 조건문** : 조건에 따라 코드 실행하거나 실행하지 않게 만들고 싶을 때 사용

- 비교 연산자를 사용한 조건식입니다. 결과가 참이면 True를, 거짓이면 False를 적어 보세요.

조건식	결과
<code>10 == 100</code>	<input type="text"/>
<code>10 != 100</code>	<input type="text"/>
<code>10 > 100</code>	<input type="text"/>
<code>10 < 100</code>	<input type="text"/>
<code>10 <= 100</code>	<input type="text"/>
<code>10 >= 100</code>	<input type="text"/>

- 다음 세 개의 예제 중 “참입니다”를 출력하는 것은 몇 번인가요?

`x = 2`
`if x > 4:`
 `print("참입니다")`

①

`x = 1`
`if x > 4:`
 `print("참입니다")`

②

`x = 10`
`if x > 4:`
 `print("참입니다")`

③

- 사용자로부터 숫자 두 개를 입력받고 첫 번째 입력받은 숫자가 큰지, 두 번째 입력받은 숫자가 큰지를 구하는 프로그램을 다음 빈칸을 채워 완성해 보세요.

```
a =  (input("> 1번째 숫자: "))
b =  (input("> 2번째 숫자: "))
print()

if  :
    print("처음 입력했던 {}가 {}보다 더 큼니다".)
if  :
    print("두 번째로 입력했던 {}가 {}보다 더 큼니다".)
```

 실행결과

> 1번째 숫자: 100

> 2번째 숫자: 10

처음 입력했던 100가 10보다 더 큼니다

2. if~else와 elif구문



[참고] 윤인성, "혼자 공부하는 파이썬", 한빛미디어

[핵심 키워드] else 구문, elif 구문, False 값, pass

[핵심 포인트] if 조건문은 뒤에 else 구문을 붙여서 사용할 수 있다. 이처럼 if 구문 뒤에 else 구문을 붙인 것을 if else 조건문이라 부르기도 한다. 이것이 어떠한 경우에 사용하는 조건문인지 알아본다.

- 정반대되는 상황에서 두 번이나 if 조건문을 사용해 조건을 비교하는 것은 낭비일 수 있다.

```
01  # 입력을 받습니다.  
02  number = input("정수 입력> ")  
03  number = int(number)  
04  
05  # 짝수 조건  
06  if number % 2 == 0:  
07      print("짝수입니다")  
08  
09  # 홀수 조건  
10  if number % 2 == 1:  
11      print("홀수입니다")
```

● else 구문

- if 조건문 뒤에 사용하며, if 조건문의 조건이 거짓일 때 실행되는 부분

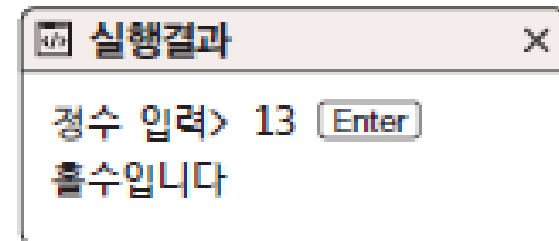
```
if 조건:  
    □□□□조건이 참일 때 실행할 문장  
else:  
    □□□□조건이 거짓일 때 실행할 문장
```

□□□□는 들여쓰기 4칸

- 조건문이 오로지 두 가지로만 구분될 때 if else 구문을 사용하면 조건 비교를 단 한번만 하므로 이전의 코드보다 두 배 효율적

➤ 예시 - if 조건문에 else 구문 추가해서 짝수와 홀수 구분

```
01  # 입력을 받습니다.  
02  number = input("정수 입력> ")  
03  number = int(number)  
04  
05  # 조건문을 사용합니다.  
06  if number % 2 == 0:  
07      # 조건이 참일 때, 즉 짝수 조건  
08      print("짝수입니다")  
09  else:  
10      # 조건이 거짓일 때, 즉 홀수 조건  
11      print("홀수입니다")
```



● elif 구문

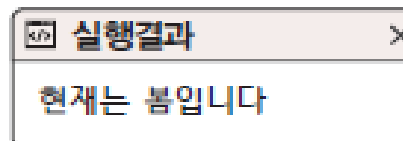
- 세 개 이상의 조건을 연결해서 사용
- if 조건문과 else 구문 사이에 입력

```
if 조건A:  
    □□□□조건A가 참일 때 실행할 문장  
elif 조건B:  
    □□□□조건B가 참일 때 실행할 문장  
elif 조건C:  
    □□□□조건C가 참일 때 실행할 문장  
...  
else:  
    □□□□모든 조건이 거짓일 때 문장
```

□□□□는 들여쓰기 4칸

➤ 예시 - 계절 구하기

```
01  # 날짜/시간과 관련된 기능을 가져옵니다.
02  import datetime
03
04  # 현재 날짜/시간을 구하고
05  # 쉽게 사용할 수 있게 월을 변수에 저장합니다.
06  now = datetime.datetime.now()
07  month = now.month
08
09  # 조건문으로 계절을 확인합니다.
10  if 3 <= month <= 5:
11      print("현재는 봄입니다.")
12  elif 6 <= month <= 8:
13      print("현재는 여름입니다.")
14  elif 9 <= month <= 11:
15      print("현재는 가을입니다.")
16  else:
17      print("현재는 겨울입니다.")
```



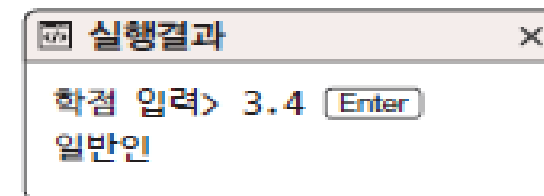
● 조건문의 활용

➤ 예시

조건	설명(학생 평가)	조건	설명(학생 평가)
4.5	신	1.75~2.3	오락문화의 선구자
4.2~4.5	교수님의 사랑	1.0~1.75	불가촉천민
3.5~4.2	현 체제의 수호자	0.5~1.0	자벌레
2.8~3.5	일반인	0~0.5	플랑크톤
2.3~2.8	일탈을 꿈꾸는 소시민	0	시대를 앞서가는 혁명의 씨앗

```
01  # 변수를 선언합니다.
02  score = float(input("학점 입력> "))
03
04  # 조건문을 적용합니다.
05  if score == 4.5:
06      print("신")
07  elif 4.2 <= score < 4.5:
08      print("교수님의 사랑")
```

```
09 elif 3.5 <= score < 4.2:
10     print("현 체제의 수호자")
11 elif 2.8 <= score < 3.5:
12     print("일반인")
13 elif 2.3 <= score < 2.8:
14     print("일탈을 꿈꾸는 소시민")
15 elif 1.75 <= score < 2.3:
16     print("오락문화의 선구자")
17 elif 1.0 <= score < 1.75:
18     print("불가촉천민")
19 elif 0.5 <= score < 1.0:
20     print("자벌레")
21 elif 0 < score < 0.5:
22     print("플랑크톤")
23 elif score == 0:
24     print("시대를 앞서가는 혁명의 씨앗")
```



- 위에서 제외된 조건을 한 번 더 검사하여 비효율적

```
01 # 변수를 선언합니다.
02 score = float(input("학점 입력> "))
03
04 # 조건문을 적용합니다.
05 if score == 4.5:
06     print("신")
07 elif 4.2 <= score:
08     print("교수님의 사랑")
09 elif 3.5 <= score:
10     print("현 체제의 수호자")
11 elif 2.8 <= score:
12     print("일반인")
13 elif 2.3 <= score:
14     print("일탈을 꿈꾸는 소시민")
15 elif 1.75 <= score:
16     print("오락문화의 선구자")
17 elif 1.0 <= score:
18     print("불가촉천민")
19 elif 0.5 <= score:
20     print("자벌레")
21 elif 0 < score:
22     print("플랑크톤")
23 else:
24     print("시대를 앞서가는 혁명의 씨앗")
```

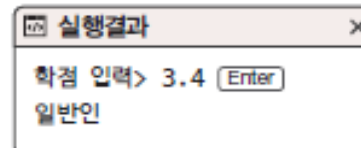
하위 값만 검사하고 상위 값은 검사를 생략

elif 4.2 <= score < 4.5:



elif 4.2 <= score:

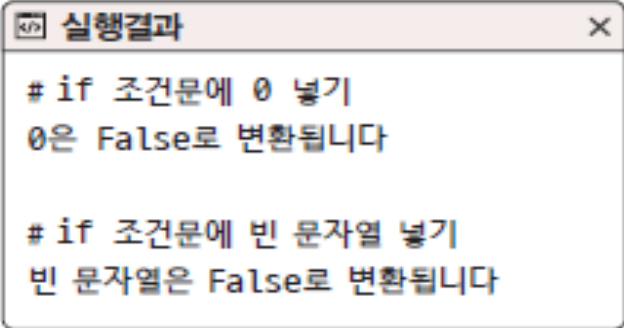
조건 비교를 반으로 줄이고 코드 가독성 향상됨



● 빈 컨테이너

- if 조건문의 매개변수에 볼 아닌 다른 값이 올 때 자동으로 불로 변환
- 이 때 False로 변환되는 값: None, 0.0, 빈 문자열, 빈 바이트열, 빈 리스트

```
01 print("# if 조건문에 0 넣기")
02 if 0:
03     print("0은 True로 변환됩니다")
04 else:
05     print("0은 False로 변환됩니다")
06 print()
07
08 print("# if 조건문에 빈 문자열 넣기")
09 if "":
10     print("빈 문자열은 True로 변환됩니다")
11 else:
12     print("빈 문자열은 False로 변환됩니다")
```



```
실행결과
# if 조건문에 0 넣기
0은 False로 변환됩니다

# if 조건문에 빈 문자열 넣기
빈 문자열은 False로 변환됩니다
```

- 나중에 구현하고자 구문을 비워 두는 경우

```
if zero == 0
    빈 줄 삽입
else:
    빈 줄 삽입
```

```
01  # 입력을 받습니다.
02  number = input("정수 입력> ")
03  number = int(number)
04
05  # 조건문 사용
06  if number > 0:
07      # 양수일 때: 아직 미구현 상태입니다.
08  else:
09      # 음수일 때: 아직 미구현 상태입니다.
```

- IndentationError

- if 조건문 사이에는 무조건 들여쓰기 4칸 넣고 코드 작성해야 함

- pass 키워드

- 아무것도 작성하지 않고 임시적으로 비워 둠

```
01  # 입력을 받습니다.  
02  number = input("정수 입력> ")  
03  number = int(number)  
04  
05  # 조건문 사용  
06  if number > 0:  
07      # 양수일 때: 아직 미구현 상태입니다.  
08      pass  
09  else:  
10      # 음수일 때: 아직 미구현 상태입니다.  
11      pass
```

- **else 구문** : if 조건문 뒤에 사용하며, if 조건문의 조건이 거짓일 때 실행
- **elif 구문** : if 조건문과 else 구문 사이에 입력하며, 세 개 이상의 조건을 연결해서 사용할 때 적절
- **False로 변환되는 값** : if 조건문의 조건식에서 False로 변환되는 값은 None, 0, 0.0, 빈 문자열, 빈 바이트 열, 빈 리스트, 빈 튜플, 빈 딕셔너리 등이 있음
- **pass 키워드** : 프로그래밍의 전체 골격을 잡아두고 내부에 처리할 내용은 나중에 만들고자 할 때 pass 키워드 입력

- 다음 코드의 실행결과를 예측해 빈칸에 결괏값을 입력하세요. 아래의 코드는 모두 같고 입력 결과가 다른 경우입니다.

```
x = 2
y = 10

if x > 4:
    if y > 2:
        print(x * y)
else:
    print(x + y)
```

```
x = 1
y = 4

if x > 4:
    if y > 2:
        print(x * y)
else:
    print(x + y)
```

```
x = 10
y = 2

if x > 4:
    if y > 2:
        print(x * y)
else:
    print(x + y)
```

- 다음 중첩 조건문에 논리 연산자 적용해 하나의 if 조건문으로 만들어 주세요.

```
if x > 10:  
    if x < 20:  
        print("조건에 맞습니다.")
```



```
print("조건에 맞습니다.")
```

3. 리스트와 반복문



[참고] 윤인성, "혼자 공부하는 파이썬", 한빛미디어

[핵심 키워드] 리스트, 요소, 인덱스, for 반복문

[핵심 포인트] 여러 개의 값을 나타낼 수 있게 해주는 리스트, 딕셔너리 등의 자료형도 존재한다. 이번 절에서는 리스트에 대해 알아보고, 이러한 자료가 반복문에 의해 어떻게 활용되는지 살펴본다.

● 리스트 (list)

- 여러 가지 자료를 저장할 수 있는 자료
- 자료들을 모아서 사용할 수 있게 해 줌
- 대괄호 내부에 자료들 넣어 선언

```
>>> array = [273, 32, 103, "문자열", True, False]
>>> print(array)
[273, 32, 103, '문자열', True, False]
```

- 요소 (element)

➤ 리스트의 대괄호 내부에 넣는 자료

```
[요소, 요소, 요소...]
```

```
>>> [1, 2, 3, 4] # 숫자만으로 구성된 리스트
[1, 2, 3, 4]
>>> ["안", "녕", "하", "세", "요"] # 문자열만으로 구성된 리스트
['안', '녕', '하', '세', '요']
>>> [273, 32, 103, "문자열", True, False] # 여러 자료형으로 구성된 리스트
[273, 32, 103, '문자열', True, False]
```

- 리스트 내부의 요소 각각 사용하려면 리스트 이름 바로 뒤에 대괄호 입력 후 자료의 위치 나타내는 숫자 입력

```
list_a = [273, 32, 103, "문자열", True, False]
```

list_a	273	32	103	문자열	True	False
	[0]	[1]	[2]	[3]	[4]	[5]

➤ 인덱스 (index)

- 대괄호 안에 들어간 숫자


```
>>> list_a = [273, 32, 103, "문자열", True, False]
>>> list_a[0]
273
>>> list_a[1]
32
>>> list_a[2]
103
>>> list_a[1:3]
[32, 103]
```

- 결과로 [32, 103] 출력

- 리스트 특정 요소를 변경할 수 있음

```
>>> list_a = [273, 32, 103, "문자열", True, False]
>>> list_a[0] = "변경"
>>> list_a
['변경', 32, 103, '문자열', True, False]
```

[0] 번째 요소가 변경되었습니다.

list_a	↑					
	변경	32	103	문자열	True	False
	[0]	[1]	[2]	[3]	[4]	[5]

- 대괄호 안에 음수 넣어 뒤에서부터 요소 선택하기

```
>>> list_a = [273, 32, 103, "문자열", True, False]
>>> list_a[-1]
False
>>> list_a[-2]
True
>>> list_a[-3]
'문자열'
```

273	32	103	문자열	True	False
[-6]	[-5]	[-4]	[-3]	[-2]	[-1]

- 리스트 접근 연산자를 이중으로 사용할 수 있음

```
>>> list_a = [273, 32, 103, "문자열", True, False]
>>> list_a[3]
'문자열'
>>> list_a[3][0]
'문'
```

- 리스트 여러 개를 가지는 리스트

```
>>> list_a = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
>>> list_a[1]
[4, 5, 6]
>>> list_a[1][1]
5
```

● 리스트에서의 IndexError 예외

- 리스트의 길이 넘는 인덱스로 요소에 접근하려는 경우 발생

```
>>> list_a = [273, 32, 103]
>>> list_a[3]
```

오류

```
Traceback (most recent call last):
  File "<pyshell#3>", line 1, in <module>
    IndexError: list index out of range
```

- 예시 - 리스트 연산자

```
01  # 리스트를 선언합니다.  
02  list_a = [1, 2, 3]  
03  list_b = [4, 5, 6]  
04  
05  # 출력합니다.  
06  print("# 리스트")  
07  print("list_a =", list_a)
```

```
08 print("list_b =", list_b)
09 print()
10
11 # 기본 연산자
12 print("# 리스트 기본 연산자")
13 print("list_a + list_b =", list_a + list_b)
14 print("list_a * 3 =", list_a * 3)
15 print()
16
17 # 함수
18 print("# 길이 구하기")
19 print("len(list_a) =", len(list_a))
```

실행결과

```
# 리스트
list_a = [1, 2, 3]
list_b = [4, 5, 6]

# 리스트 기본 연산자
list_a + list_b = [1, 2, 3, 4, 5, 6]
list_a * 3 = [1, 2, 3, 1, 2, 3, 1, 2, 3]

# 길이 구하기
len(list_a) = 3
```

- 13행에서 문자열 연결 연산자 사용해 2, 3행과 7, 8행에서 선언 및 출력된 list_a와 list_b의 자료 연결
- 14행에서 문자열 반복 연산자 사용해 list_a의 자료 3번 반복
- 19행에서 len() 함수로 list_a에 들어있는 요소의 개수 구함

- append() 함수

- 리스트 뒤에 요소를 추가

```
리스트명.append(요소)
```

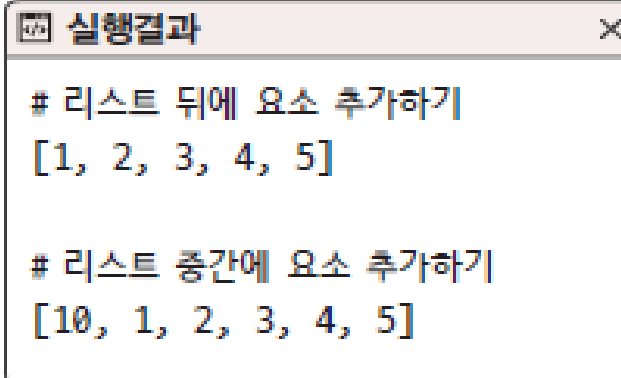
- insert() 함수

- 리스트 중간에 요소를 추가

```
리스트명.insert(위치, 요소)
```

● 예시

```
01  # 리스트를 선언합니다.
02  list_a = [1, 2, 3]
03
04  # 리스트 뒤에 요소 추가하기
05  print("# 리스트 뒤에 요소 추가하기")
06  list_a.append(4)
07  list_a.append(5)
08  print(list_a)
09  print()
10
11  # 리스트 중간에 요소 추가하기
12  print("# 리스트 중간에 요소 추가하기")
13  list_a.insert(0, 10)
14  print(list_a)
```



실행결과

```
# 리스트 뒤에 요소 추가하기
[1, 2, 3, 4, 5]

# 리스트 중간에 요소 추가하기
[10, 1, 2, 3, 4, 5]
```

➤ 6 및 7행 실행 결과

```
list_a.append(4)
list_a.append(5)
```

1	2	3	4	5
[0]	[1]	[2]	[3]	[4]

➤ 13행 실행 결과

```
list_a.insert(0, 10)
```

삽입할 값
↓
삽입할 위치

10	1	2	3	4	5
[0]	[1]	[2]	[3]	[4]	[5]

↓
여번째 위치에 10을 추가합니다.

● extend() 함수

- 원래 리스트 뒤에 새로운 리스트의 요소 모두 추가
- 매개변수로 리스트 입력

```
>>> list_a = [1, 2, 3]
>>> list_a.extend([4, 5, 6])
>>> print(list_a)
[1, 2, 3, 4, 5, 6]
```

- 리스트 연결 연산자와 요소 추가의 차이

- 리스트 연결 연산자 사용하면 결과상 원본에 변화는 없음

```
>>> list_a = [1, 2, 3]
>>> list_b = [4, 5, 6]
>>> list_a + list_b → 리스트 연결 연산자로 연결하니,
[1, 2, 3, 4, 5, 6] → 실행결과로 [1, 2, 3, 4, 5, 6]이 나왔습니다.
>>> list_a → list_a와 list_b에는 어떠한 변화도 없습니다(비파괴적 처리).
[1, 2, 3]
>>> list_b
[4, 5, 6]
```

➤ extend() 함수 사용할 경우

```
>>> list_a = [1, 2, 3]
>>> list_b = [4, 5, 6]
>>> list_a.extend(list_b) → 실행결과로 아무 것도 출력하지 않았습니다.
>>> list_a → 앞에 입력했던 list_a 자체에 직접적인 변화가 있습니다(파괴적 처리).
[1, 2, 3, 4, 5, 6]
>>> list_b
[4, 5, 6]
```

➤ 파괴적 / 비파괴적

- 인덱스로 제거하기: del 키워드, pop() 함수

```
del 리스트명[인덱스]
```

```
리스트명.pop(인덱스)
```

```
01 list_a = [0, 1, 2, 3, 4, 5]
02 print("# 리스트의 요소 하나 제거하기")
03
04 # 제거 방법[1] - del
05 del list_a[1]
06 print("del list_a[1]:", list_a)
07
08 # 제거 방법[2] - pop()
09 list_a.pop(2)
10 print("pop(2):", list_a)
```

실행결과

```
# 리스트의 요소 하나 제거하기
del list_a[1]: [0, 2, 3, 4, 5]
pop(2): [0, 2, 4, 5]
```

- 5행 실행하면 자료에서 1 제거

```
del list_a[1]
```

0	2	3	4	5
[0]	[1]	[2]	[3]	[4]

- 9행에서 2번째 요소인 3 제거

```
list_a.pop(2)
```

0	2	4	5
[0]	[1]	[2]	[3]

- **del 키워드** 사용할 경우 범위 지정해 리스트 요소를 한꺼번에 제거 가능

```
>>> list_b = [0, 1, 2, 3, 4, 5, 6]
>>> del list_b[3:6]
>>> list_b
[0, 1, 2, 6]
```

- 범위 한 쪽을 입력하지 않으면 지정 위치 기준으로 한쪽을 전부 제거

```
>>> list_c = [0, 1, 2, 3, 4, 5, 6]
>>> del list_c[:3]
>>> list_c
[3, 4, 5, 6]
```

- 값으로 제거하기: `remove()` 함수

- 특정 값을 지정하여 제거

```
리스트.remove(값)
```

```
>>> list_c = [1, 2, 1, 2]    # 리스트 선언하기
>>> list_c.remove(2)         # 리스트의 요소를 값으로 제거하기
>>> list_c
[1, 1, 2]
```

- 모두 제거하기 : `clear()` 함수

- 리스트 내부의 요소를 모두 제거

```
리스트.clear()
```

```
>>> list_d = [0, 1, 2, 3, 4, 5]
>>> list_d.clear()
>>> list_d
[] → 요소가 모두 제거되었습니다.
```

● in 연산자

- 특정 값이 리스트 내부에 있는지 확인

값 in 리스트

```
>>> list_a = [273, 32, 103, 57, 52]
>>> 273 in list_a
True
>>> 99 in list_a
False
>>> 100 in list_a
False
>>> 52 in list_a
True
```

● not in 연산자

➤ 리스트 내부에 해당 값이 없는지 확인

```
>>> list_a = [273, 32, 103, 57, 52]
>>> 273 not in list_a
False
>>> 99 not in list_a
True
>>> 100 not in list_a
True
>>> 52 not in list_a
False
>>> not 273 in list_a
False
```

● 반복문

➤ 컴퓨터에 반복 작업을 지시

```
print("출력")  
print("출력")  
print("출력")  
print("출력")  
print("출력")
```

```
for i in range(100):  
    print("출력")
```

→ 반복에 사용할 수 있는 자료

- 문자열, 리스트, 딕셔너리 등과 조합하여 for 반복문을 사용

for 반복자 in 반복할 수 있는 것:

코드

```
01  # 리스트를 선언합니다.  
02  array = [273, 32, 103, 57, 52]  
03  
04  # 리스트에 반복문을 적용합니다.  
05  for element in array:  
06      # 출력합니다.  
07      print(element)
```

실행결과

273
32
103
57
52

- **리스트** : 여러 가지 자료를 저장할 수 있는 자료형
- **요소** : 리스트 내부에 있는 각각의 내용을 의미
- **인덱스** : 리스트 내부에서 값의 위치를 의미
- **for 반복문** : 특정 코드를 반복해서 실행할 때 사용하는 기본 구문

- list_a = [0, 1, 2, 3, 4, 5, 6, 7] 입니다. 다음 표의 함수들을 실행했을 때 list_a의 결과가 어떻게 나오는지 적어보세요

함수	list_a의 값
list_a.extend(list_a)	
list_a.append(10)	
list_a.insert(3, 0)	
list_a.remove(3)	
list_a.pop(3)	
list_a.clear()	

- 다음 반복문 내부에 if 조건문의 조건식을 채워서 100 이상의 숫자만 출력하게 만들어보세요.

```
numbers = [273, 103, 5, 32, 65, 9, 72, 800, 99]

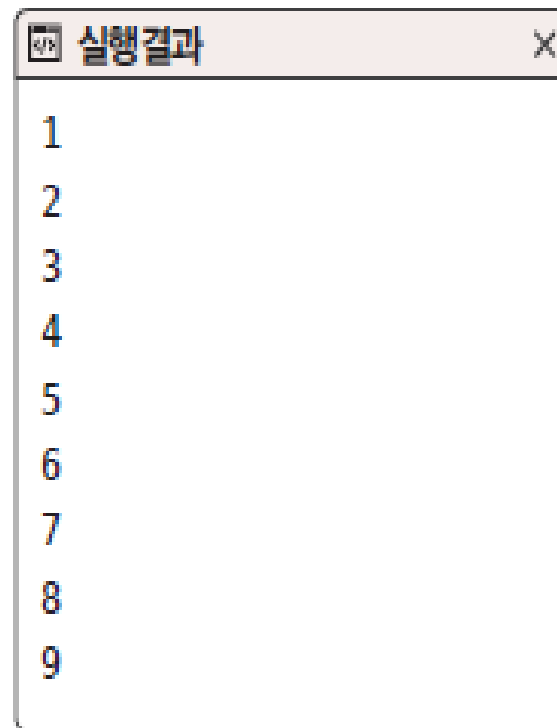
for number in numbers:
    if :
        print("- 100 이상의 수:", number)
```

실행결과

- 100 이상의 수: 273
- 100 이상의 수: 103
- 100 이상의 수: 800

- 다음 빈칸을 채워서 실행결과처럼 숫자를 하나하나 모두 출력해보세요

```
list_of_list = [  
    [1, 2, 3],  
    [4, 5, 6, 7],  
    [8, 9],  
]
```



4. 딕셔너리와 반복문



[참고] 윤인성, "혼자 공부하는 파이썬", 한빛미디어

[핵심 키워드] : 딕셔너리, 키, 값

[핵심 포인트]

여러 개의 값을 나타낼 수 있게 해주는 자료형 중
딕셔너리에 대해 알아봅니다.

● 딕셔너리 (dictionary)

➤ 키를 기반으로 값을 저장하는 것

```
{
  키      값
  ↓      ↓
  "키A": 10,      # 문자열을 키로 사용하기
  "키B": 20,
  "키C": 30,
  1:      40,      # 숫자를 키로 사용하기
  False: 50        # 불을 키로 사용하기
}
```

자료형	의미	가리키는 위치	선언 형식
리스트	인덱스를 기반으로 값을 저장	인덱스	변수 = []
딕셔너리	키를 기반으로 값을 저장	키	변수 = {}

● 딕셔너리 선언

- 중괄호로 선언하며 '키: 값' 형태를 쉼표로 연결해서 만듦

```
변수 = {  
    키: 값,  
    키: 값,  
    ...  
    키: 값  
}
```

```
>>> dict_a = {  
    "name": "어벤저스 엔드게임",  
    "type": "히어로 무비"  
}
```

- 특정 키 값만 따로 출력하기

- 딕셔너리 뒤에 대괄호 입력하고 그 내부에 키 입력

```
>>> dict_a  
{'name': '어벤저스 엔드게임', 'type': '히어로 무비'}
```

```
>>> dict_a["name"]  
'어벤저스 엔드게임'  
>>> dict_a["type"]  
'히어로 무비'
```


- 딕셔너리 내부 값에 문자열, 숫자, 불 등 다양한 자료 넣기

```
>>> dict_b = {  
    "director": ["안소니 루소", "조 루소"],  
    "cast": ["아이언맨", "타노스", "토르", "닥터스트레인지", "헐크"]  
}
```

```
>>> dict_b  
{'director': ['안소니 루소', '조 루소'], 'cast': ['아이언맨', '타노스', '토르', '닥터스트레인지', '헐크']}  
>>> dict_b["director"]  
['안소니 루소', '조 루소']
```

구분	선언 형식	사용 예	불린 예
리스트	<code>list_a = []</code>	<code>list_a[1]</code>	
딕셔너리	<code>dict_a = {}</code>	<code>dict_a["name"]</code>	<code>dict_a{"name"}</code>

➤ 예시 - 딕셔너리 요소에 접근하기

```
01  # 딕셔너리를 선언합니다.
02  dictionary = {
03      "name": "7D 건조 망고",
04      "type": "당절임",
05      "ingredient": ["망고", "설탕", "메타중아황산나트륨", "치자황색소"],
06      "origin": "필리핀"
07  }
08
```

```
09  # 출력합니다.  
10  print("name:", dictionary["name"])  
11  print("type:", dictionary["type"])  
12  print("ingredient:", dictionary["ingredient"])  
13  print("origin:", dictionary["origin"])  
14  print()  
15  
16  # 값을 변경합니다.  
17  dictionary["name"] = "8D 건조 망고"  
18  print("name:", dictionary["name"])
```

실행결과

```
name: 7D 건조 망고  
type: 당결임  
ingredient: ['망고', '설탕', '메타중아황산나트륨', '치자!  
origin: 필리핀  
  
name: 8D 건조 망고
```

- 리스트 안의 특정 값 출력하려는 경우

```
>>> dictionary["ingredient"]  
['망고', '설탕', '메타중아황산나트륨', '치자황색소']  
>>> dictionary["ingredient"][1]  
'설탕'
```

- 딕셔너리의 문자열 키와 관련된 실수
- **NameError 오류**

➤ name이라는 이름이 정의되지 않음

```
>>> dict_key = {  
    name: "7D 건조 망고",  
    type: "당절임"  
}
```

 오류

```
Traceback (most recent call last):  
  File "<pyshell#5>", line 2, in <module>  
    name: "7D 건조 망고",  
NameError: name 'name' is not defined
```

- name 이름을 변수로 만들어 해결

```
>>> name = "이름"
>>> dict_key = {
    name: "7D 건조 망고",
    type: "당절임"
}
>>> dict_key
{'이름': '7D 건조 망고', <class 'type': '당절임'}
```

- 딕셔너리에 값 추가할 때는 키를 기반으로 값 입력

딕셔너리[새로운 키] = 새로운 값

- 슬라이드 #8, 9에서 만든 dictionary에 새로운 자료 추가

```
>>> dictionary["price"] = 5000
>>> dictionary
{'name': '8D 건조 망고', 'type': '당절임', 'ingredient': ['망고', '설탕', '메타중아황산나트륨', '치자황색소'], 'origin': '필리핀', 'price': 5000}
```

→ "price" 키가 추가되었습니다.

- 딕셔너리에 이미 존재하는 키 지정하고 값 넣으면 기존 값을 대치

```
>>> dictionary["name"] = "8D 건조 파인애플"
>>> dictionary
{'name': '8D 건조 파인애플', 'type': '당절임', 'ingredient': ['망고', '설탕', '메타중아황산나트륨', '치자황색소'], 'origin': '필리핀', 'price': 5000}
```

새로운 값으로 대치되었습니다.

- 딕셔너리 요소의 제거 : del 키워드

```
>>> del dictionary["ingredient"]  
>>> dictionary  
{'name': '8D 건조 파인애플', 'type': '당절임', 'origin': '필리핀', 'price': 5000}
```


➤ 예시 - 딕셔너리에 요소 추가하기

```
01  # 딕셔너리를 선언합니다.
02  dictionary = {}
03
04  # 요소 추가 전에 내용을 출력해 봅니다.
05  print("요소 추가 이전:", dictionary)
06
07  # 딕셔너리에 요소를 추가합니다.
08  dictionary["name"] = "새로운 이름"
09  dictionary["head"] = "새로운 정신"
10  dictionary["body"] = "새로운 몸"
11
12  # 출력합니다.
13  print("요소 추가 이후:", dictionary)
```

실행결과

요소 추가 이전: {}

요소 추가 이후: {'name': '새로운 이름', 'head': '새로운 정신', 'body': '새로운 몸'}

➤ 예시 - 딕셔너리에 요소 제거하기

```
01  # 딕셔너리를 선언합니다.
02  dictionary = {
03      "name": "7D 건조 망고",
04      "type": "당절임"
05  }
06
07  # 요소 제거 전에 내용을 출력해 봅니다.
08  print("요소 제거 이전:", dictionary)
09
10  # 딕셔너리의 요소를 제거합니다.
11  del dictionary["name"]
12  del dictionary["type"]
13
14  # 요소 제거 후에 내용을 출력해 봅니다.
15  print("요소 제거 이후:", dictionary)
```

실행결과

요소 제거 이전: {'name': '7D 건조 망고', 'type': '당절임'}
요소 제거 이후: {}

● KeyError 예외

- 딕셔너리에서 존재하지 않는 키에 접근할 경우

```
>>> dictionary = {}  
>>> dictionary["Key"]
```

오류

```
Traceback (most recent call last):  
  File "<pyshell#7>", line 1, in <module>  
    dictionary["Key"]  
KeyError: 'Key'
```

- 값 제거할 경우도 같은 원리

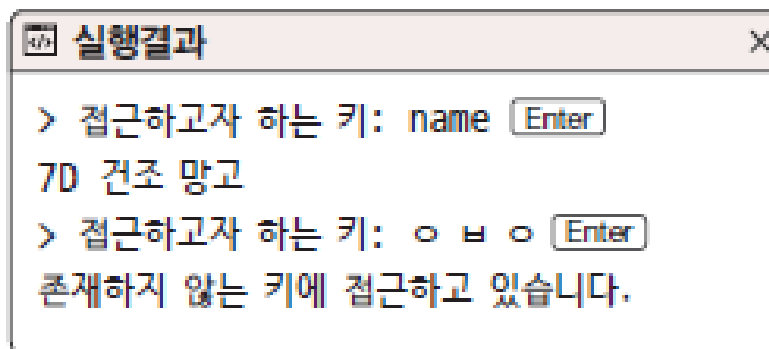
```
>>> del dictionary["Key"]  
Traceback (most recent call last):  
  File "<pyshell#8>", line 1, in <module>  
    del dictionary["Key"]  
KeyError: 'Key'
```

● in 키워드

- 사용자로부터 접근하고자 하는 키 입력 받은 후 존재하는 경우에만 접근하여 값을 출력

```
01  # 딕셔너리를 선언합니다.
02  dictionary = {
03      "name": "7D 건조 망고",
04      "type": "당절임",
05      "ingredient": ["망고", "설탕", "메타중아황산나트륨", "치자황색소"],
06      "origin": "필리핀"
07  }
08
09  # 사용자로부터 입력을 받습니다.
10  key = input("> 접근하고자 하는 키: ")
```


```
11
12 # 출력합니다.
13 if key in dictionary:
14     print(dictionary[key])
15 else:
16     print("존재하지 않는 키에 접근하고 있습니다.")
```



● get() 함수

- 딕셔너리의 키로 값을 추출, 존재하지 않는 키에 접근할 경우 None 출력

```
01 # 딕셔너리를 선언합니다.
02 dictionary = {
03     "name": "7D 건조 망고",
04     "type": "당절임",
05     "ingredient": ["망고", "설탕", "메타중아황산나트륨", "치자황색소"],
06     "origin": "필리핀"
07 }
08
09 # 존재하지 않는 키에 접근해 봅니다.
10 value = dictionary.get("존재하지 않는 키")
11 print("값:", value)
12
13 # None 확인 방법
14 if value == None: → None과 같은지 확인만 하면 됩니다.
15     print("존재하지 않는 키에 접근했었습니다.")
```

 실행결과

값: None
존재하지 않는 키에 접근했었습니다.

- for 반복문과 딕셔너리의 조합

```
for 키 변수 in 딕셔너리:  
    코드
```



```
01  # 딕셔너리를 선언합니다.
02  dictionary = {
03      "name": "7D 건조 망고",
04      "type": "당절임",
05      "ingredient": ["망고", "설탕", "메타중아황산나트륨", "치자황색소"],
06      "origin": "필리핀"
07  }
08
09  # for 반복문을 사용합니다.
10  for key in dictionary:
11      # 출력합니다.
12      print(key, ":", dictionary[key])
```

실행결과

```
name : 7D 건조 망고
type : 당절임
ingredient : ['망고', '설탕', '메타중아황산나트륨', '치자황색소']
origin : 필리핀
```

- **딕셔너리** : 키를 기반으로 여러 자료 저장하는 자료형
- **키** : 딕셔너리 내부에서 값에 접근할 때 사용하는 것
- **값** : 딕셔너리 내부에 있는 각각의 내용

- 다음 표에서 dict_a의 결과가 나오도록 빈칸을 채워보세요.

dict_a의 값	dict_a에 적용할 코드	dict_a의 결과
<code>{}</code>	<input type="text"/>	<code>{ "name": "구름" }</code>
<code>{ "name": "구름" }</code>	<input type="text"/>	<code>{}</code>

- 다음 빈칸을 채워서 numbers 내부에 들어있는 숫자가 몇 번 등장하는지를 출력하는 코드를 작성해보세요.

```
# 숫자는 무작위로 입력해도 상관 없습니다.  
numbers = [1,2,6,8,4,3,2,1,9,5,4,9,7,2,1,3,5,4,8,9,7,2,3]  
counter = {}  
  
for number in numbers:  
  
  
  
  
  
  
  
  
# 최종 출력  
print(counter)
```

실행결과

{1: 3, 2: 4, 6: 1, 8: 2, 4: 3, 3: 3, 9: 3, 5: 2, 7: 2}

- 아래 예시를 참조해 다음 빈칸을 채워 실행결과와 같이 출력되게 만들어보세요.

```
type("문자열") is str # 문자열인지 확인  
type([]) is list      # 리스트인지 확인  
type({}) is dict      # 딕셔너리인지 확인
```

딕셔너리를 선언합니다.

```
character = {  
    "name": "기사",  
    "level": 12,  
    "items": {  
        "sword": "불꽃의 검",  
        "armor": "풀플레이트"  
    },  
    "skill": ["베기", "세게 베기", "아주 세게 베기"]  
}
```

for 반복문을 사용합니다.

```
for key in character:
```

실행결과

```
name : 기사  
level : 12  
sword : 불꽃의 검  
armor : 풀플레이트  
skill : 베기  
skill : 세게 베기  
skill : 아주 세게 베기
```

5. 반복문과 while 반복문



[참고] 윤인성, "혼자 공부하는 파이썬", 한빛미디어

[핵심 키워드] : 범위, while 반복문, break 키워드, continue 키워드

[핵심 포인트]

특정 횟수 / 특정 시간만큼, 그리고 어떤 조건이 될 때까지 반복하는 등의 경우에 대해 알아본다.

- 범위 (range)

- 특정 횟수만큼 반복해서 돌리고 싶을 때 for 반복문과 조합하여 사용

- 매개변수에 숫자를 한 개 넣는 방법

- 0부터 A-1까지의 정수로 범위 만들기

`range(A)` → A는 숫자

- 매개변수에 숫자를 두 개 넣는 방법

- A부터 B-1까지의 정수로 범위 만들기

`range(A, B)` → A와 B는 숫자

- 매개변수에 숫자를 세 개 넣는 방법

- A부터 B-1까지의 정수로 범위 만들기 앞뒤의 숫자가 C만큼의 차이 가짐

`range(A, B, C)` → A, B, C는 숫자

➤ 예시

- 매개변수에 숫자 한 개 넣은 범위

```
>>> a = range(5)
```

```
>>> a  
range(0, 5)
```

```
>>> list(range(10))  
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

- 매개변수에 숫자 두 개 넣은 범위

```
>>> list(range(0, 5)) → 0부터 (5-1)까지의 정수로 범위를 만듭니다.
```

```
[0, 1, 2, 3, 4]
```

```
>>> list(range(5, 10)) → 5부터 (10-1)까지의 정수로 범위를 만듭니다.
```

```
[5, 6, 7, 8, 9]
```

- 매개변수에 숫자 세 개 넣은 범위

```
>>> list(range(0, 10, 2)) → 0부터 2씩 증가하면서 (10-1)까지의 정수로 범위를 만듭니다.
```

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

```
>>> list(range(0, 10, 3)) → 0부터 3씩 증가하면서 (10-1)까지의 정수로 범위를 만듭니다.
```

```
[0, 3, 6, 9]
```

- 범위 만들 때 매개변수 내부에 수식 사용하는 경우
 - 코드 특정 부분의 강조

```
>>> a = range(0, 10 + 1)
>>> list(a)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

- 예시 - 나누기 연산자 사용

```
>>> n = 10
>>> a = range(0, n / 2) → 매개변수로 나눗셈을 사용한 경우 오류가 발생합니다.
Traceback (most recent call last):
  File "<pyshell#10>", line 1, in <module>
TypeError: 'float' object cannot be interpreted as an integer
```

- TypeError 발생

- 정수 나누기 연산자

```
>>> a = range(0, int(n / 2)) → 실수를 정수로 바꾸는 방법보다
```

```
>>> list(a)
```

```
[0, 1, 2, 3, 4]
```

```
>>> a = range(0, n // 2) → 정수 나누기 연산자를 많이 사용합니다!
```

```
>>> list(a)
```

```
[0, 1, 2, 3, 4]
```

● for 반복문과 범위의 조합

for 숫자 변수 in 범위:

코드

```
01  # for 반복문과 범위를 함께 조합해서 사용합니다.
02  for i in range(5):
03      print(str(i) + "= 반복 변수")
04  print()
05
06  for i in range(5, 10):
07      print(str(i) + "= 반복 변수")
08  print()
09
10  for i in range(0, 10, 3):
11      print(str(i) + "= 반복 변수")
12  print()
```

실행결과

0 = 반복 변수
1 = 반복 변수
2 = 반복 변수
3 = 반복 변수
4 = 반복 변수

5 = 반복 변수
6 = 반복 변수
7 = 반복 변수
8 = 반복 변수
9 = 반복 변수

0 = 반복 변수
3 = 반복 변수
6 = 반복 변수
9 = 반복 변수

- 몇 번 반복인지를 알아야 하는 경우

```
# 리스트를 선언합니다.  
array = [273, 32, 103, 57, 52]  
  
# 리스트에 반복문을 적용합니다.  
for element in array:  
    # 출력합니다.  
    print(element)
```

현재 무엇을 출력하고 있는지 보다, 몇 번째 출력인지를 알아야 하는 경우가 있습니다.

```
01 # 리스트를 선언합니다.  
02 array = [273, 32, 103, 57, 52]  
03  
04 # 리스트에 반복문을 적용합니다.  
05 for i in range(len(array)):  
06     # 출력합니다.  
07     print("{}번째 반복: {}".format(i, array[i]))
```

실행결과	
0번째 반복:	273
1번째 반복:	32
2번째 반복:	103
3번째 반복:	57
4번째 반복:	52

● 역반복문

- 큰 숫자에서 작은 숫자로 반복문 적용
- `range()` 함수의 매개변수 세 개 사용하는 방법

```
01  # 역반복문
02  for i in range(4, 0 - 1, -1):
03      # 출력합니다.
04      print("현재 반복 변수: {}".format(i))
```

실행결과

현재 반복 변수: 4
현재 반복 변수: 3
현재 반복 변수: 2
현재 반복 변수: 1
현재 반복 변수: 0

➤ reversed() 함수 사용하는 방법

```
01  # 역반복문
02  for i in reversed(range(5)):
03      # 출력합니다.
04      print("현재 반복 변수: {}".format(i))
```

실행결과

현재 반복 변수: 4
현재 반복 변수: 3
현재 반복 변수: 2
현재 반복 변수: 1
현재 반복 변수: 0

● while 반복문

- 리스트 또는 딕셔너리 내부의 요소를 특정 횟수만큼 반복

while 불 표현식:
 문장

```
01  # while 반복문을 사용합니다.  
02  while True:  
03      # "."을 출력합니다.  
04      # 기본적으로 end가 "\n"이라 줄바꿈이 일어나는데  
05      # 빈 문자열 ""로 바꿔서 줄바꿈이 일어나지 않게 합니다.  
06      print(".", end="")
```

실행결과

```
.....  
.....  
.....  
.....  
.....  
.....  
.....  
.....  
.....
```

```
01  # 반복 변수를 기반으로 반복하기
02  i = 0
03  while i < 10:
04      print("{}번째 반복입니다.".format(i))
05      i += 1
```

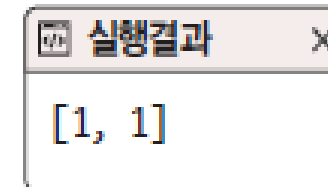
실행결과

0번째 반복입니다.
1번째 반복입니다.
2번째 반복입니다.
3번째 반복입니다.
4번째 반복입니다.
5번째 반복입니다.
6번째 반복입니다.
7번째 반복입니다.
8번째 반복입니다.
9번째 반복입니다.

- 리스트 내부에서 해당하는 값을 여러 개 제거

➤ while 반복문의 조건을 '리스트 내부에 요소가 있는 동안'으로 지정

```
01  # 변수를 선언합니다.  
02  list_test = [1, 2, 1, 2]  
03  value = 2  
04  
05  # list_test 내부에 value가 있다면 반복  
06  while value in list_test:  
07      list_test.remove(value)  
08  
09  # 출력합니다.  
10  print(list_test)
```



실행결과

[1, 1]

- 예시 - 유닉스 타임 구하기

- 시간 관련된 기능 가져오기

```
>>> import time
```

- 유닉스 타임

```
>>> time.time()  
1557241486.6654928
```

● 유닉스 타임과 while 반복문을 조합

➤ 5초 동안 반복하기

```
01  # 시간과 관련된 기능을 가져옵니다.  
02  import time  
03  
04  # 변수를 선언합니다.  
05  number = 0  
06  
07  # 5초 동안 반복합니다.  
08  target_tick = time.time() + 5  
09  while time.time() < target_tick:  
10      number += 1  
11  
12  # 출력합니다.  
13  print("5초 동안 {}번 반복했습니다.".format(number))
```

실행결과

5초 동안 14223967번 반복했습니다.

● break 키워드

➤ 반복문 벗어날 때 사용하는 키워드

```
01  # 변수를 선언합니다.  
02  i = 0  
03  
04  # 무한 반복합니다.  
05  while True:  
06      # 몇 번째 반복인지 출력합니다.  
07      print("{}번째 반복문입니다.".format(i))  
08      i = i + 1  
09      # 반복을 종료합니다.  
10      input_text = input("> 종료하시겠습니까?(y/n): ")  
11      if input_text in ["y", "Y"]:  
12          print("반복을 종료합니다.")  
13          break
```

실행결과

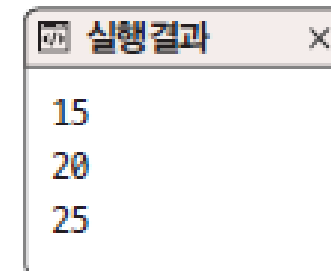
0번째 반복문입니다
> 종료하시겠습니까?(y/n): n Enter
1번째 반복문입니다
> 종료하시겠습니까?(y/n): n Enter
2번째 반복문입니다
> 종료하시겠습니까?(y/n): n Enter
3번째 반복문입니다
> 종료하시겠습니까?(y/n): n Enter
4번째 반복문입니다
> 종료하시겠습니까?(y/n): y Enter
반복을 종료합니다.

실행결과
0번째 반
> 종료하

● continue 키워드

- 현재 반복을 생략하고 다음 반복으로 넘어감

```
01  # 변수를 선언합니다.  
02  numbers = [5, 15, 6, 20, 7, 25]  
03  
04  # 반복을 돌립니다.  
05  for number in numbers:  
06      # number가 10보다 작으면 다음 반복으로 넘어갑니다.  
07      if number < 10:  
08          continue  
09      # 출력합니다.  
10      print(number)
```



실행결과

15
20
25

- if else 구문 사용도 가능한 경우이나, continue 키워드 사용하면 이후 처리의 들여쓰기를 하나 줄일 수 있음

continue 키워드를 사용하지 않은 경우

```
# 반복을 돌립니다.  
for number in numbers:  
    # 반복 대상을 한정합니다.  
    if number >= 10:  
        # 문장  
        # 문장  
        # 문장  
        # 문장  
        # 문장
```

continue 키워드를 사용한 경우

```
# 반복을 돌립니다.  
for number in numbers:  
    # 반복 대상에서 제외해버립니다.  
    if number < 10:  
        continue  
    # 문장  
    # 문장  
    # 문장  
    # 문장  
    # 문장
```

- **범위** : 정수의 범위 나타내는 값으로, range() 함수로 생성
- **while 반복문** : 조건식을 기반으로 특정 코드를 반복해서 실행할 때 사용하는 구문
- **break 키워드** : 반복문을 벗어날 때 사용하는 구문
- **continue 키워드** : 반복문의 현재 반복을 생략할 때 사용하는 구문

- 다음 표를 채워 보세요.

코드가 여러 개 나올 수 있는 경우 가장 간단한 형태를 넣어 주세요.

코드	나타내는 값
<code>range(5)</code>	<code>[0, 1, 2, 3, 4]</code>
<code>range(4, 6)</code>	
<code>range(7, 0, -1)</code>	
<code>range(3, 8)</code>	<code>[3, 4, 5, 6, 7]</code>
	<code>[3, 6, 9]</code>

- 빈칸을 채워 키와 값으로 이루어진 각 리스트를 조합해 하나의 딕셔너리를 만들어 보세요.

```
# 숫자는 무작위로 입력해도 상관없습니다.  
key_list = ["name", "hp", "mp", "level"]  
value_list = ["기사", 200, 30, 5]  
character = {}
```

```
# 최종 출력  
print(character)
```

실행결과

```
{'name': '기사', 'hp': 200, 'mp': 30, 'level': 5}
```

- 1부터 숫자를 하나씩 증가시키면서 더하는 경우를 생각해 봅시다. 몇을 더할 때 1000을 넘는지 구해 보세요. 그리고 그때의 값도 출력해보세요. 다음은 1000이 넘는 경우를 구한 예입니다.

1, $1 + 2 = 3$, $1 + 2 + 3 = 6$, $1 + 2 + 3 + 4 = 10...$

```
limit = 10000
i = 1
# sum은 파이썬 내부에서 사용하는 식별자이므로 sum_value라는 변수 이름을 사용합니다.

print("{}를 더할 때 {}을 넘으며 그때의 값은 {}입니다.".format(i, limit, sum_value))
```

실행결과

142를 더할 때 10000을 넘으며 그때의 값은 10011입니다.