

1-1

자주 나오는 파이썬 용어들

목차

- 시작하기 전에
- 표현식과 문장
- 키워드
- 식별자
- 주석
- 연산자와 자료형
- 출력 : `print()`
- 키워드로 정리하는 핵심 포인트

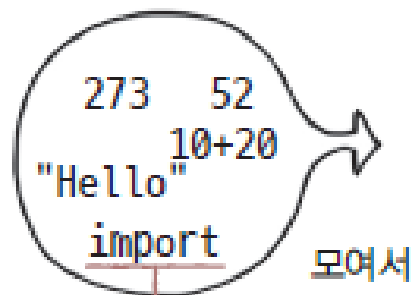
시작하기 전에

[핵심 키워드] 표현식, 키워드, 식별자, 주석, print()

[핵심 포인트] 파이썬에서 사용하는 기본 용어들은 무엇이 있는지 살펴본다.

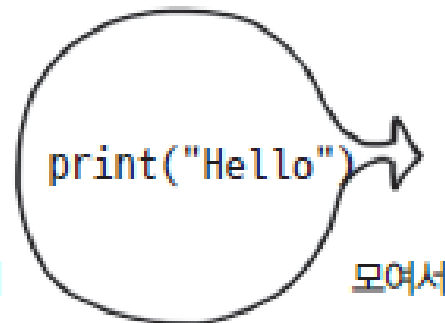
시작하기 전에

표현식
값을 만들어 내는 코드

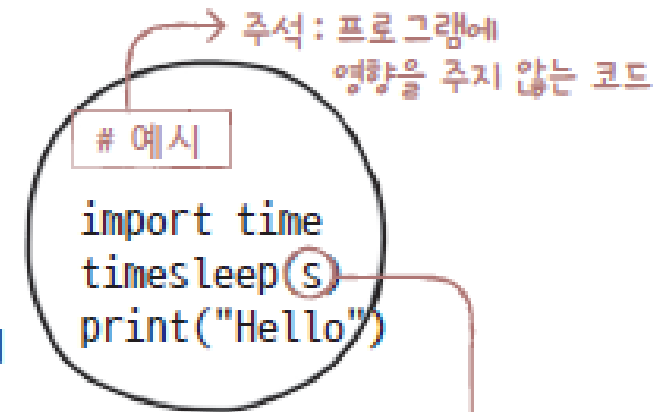


키워드 : 언어가 처음 만들어질 때 정한 단어

문장
표현식이 하나 이상 모인 것



프로그램
문장이 모인 것



식별자 : 사용자가 만들 수 있는 단어

표현식과 문장

- 표현식 (expression)
 - 파이썬에서 어떠한 값을 만들어내는 간단한 코드
 - 값이란 숫자 수식, 문자열 등이 될 수 있음

```
273
```

```
10 + 20 + 30 * 10
```

```
"Python Programming"
```

```
print("Python Programming")
```

표현식과 문장

- **문장** (statement)
 - 표현식이 하나 이상 모일 경우
 - 그 자체로 어떠한 값을 만들 수 없으면 문장이 아님

+

-

- **프로그램** (program)
 - 문장이 모여서 형성

키워드

- 키워드 (keyword)

- 특별한 의미가 부여된 단어
- 파이썬에서 이미 특정 의미로 사용하기로 예약해 놓은 것
- 프로그래밍 언어에서 이름 정할 때 똑같이 사용할 수 없음

False	None	True	and	as	assert
break	class	continue	def	del	elif
else	except	finally	for	from	global
if	import	in	is	lambda	nonlocal
not	or	pass	raise	return	try
while	with	yield			

- 대소문자 구별

키워드

- 아래 코드로 특정 단어가 파이썬 키워드인지 확인 가능

```
>>> import keyword  
>>> print(keyword.kwlist)
```

```
['False', 'None', 'True', 'and', 'as', 'assert', 'async', 'await', 'break',  
'class', 'continue', 'def', 'del', 'elif', 'else', 'except', 'finally', 'for',  
'from', 'global', 'if', 'import', 'in', 'is', 'lambda', 'nonlocal', 'not', 'or',  
'pass', 'raise', 'return', 'try', 'while', 'with', 'yield']
```


식별자

- 식별자 (identifier)

- 프로그래밍 언어에서 이름 붙일 때 사용하는 단어
- 변수 또는 함수 이름 등으로 사용
- 키워드 사용 불가
- 특수문자는 언더바(_)만 허용
- 숫자로 시작 불가
- 공백 포함 불가
- 알파벳 사용이 관례
- 의미 있는 단어로 할 것

사용 가능한 단어	사용 불가능한 단어
alpha	break → 키워드라서 안 됩니다.
alpha10	
_alpha	273alpha → 숫자로 시작해서 안 됩니다.
AlpHa	
ALPHA	has space → 공백을 포함해서 안 됩니다.

식별자

- 스네이크 케이스와 캐멀 케이스

`itemlist`

`loginstatus`

`characterhp`

`rotateangle`

- 공백이 없어 이해하기 어려움

- 스네이크 케이스 (snake case) : 언더바(_)를 기호 중간에 붙이기

- 캐멀 케이스 (camel case) : 단어들의 첫 글자를 대문자로 만들기

식별자에 공백이 없는 경우	단어 사이에 _ 기호를 붙인 경우 (스네이크 케이스)	단어 첫 글자를 대문자로 만든 경우 (캐멀 케이스)
<code>itemlist</code> <code>loginstatus</code> <code>characterhp</code> <code>rotateangle</code>	<code>item_list</code> <code>login_status</code> <code>character_hp</code> <code>rotate_angle</code>	<code>ItemList</code> <code>LoginStatus</code> <code>CharacterHp</code> <code>RotateAngle</code>

- 파이썬에서는 스네이크 및 캐멀 케이스 둘 모두 사용

식별자

- 식별자 구분하기

- 캐멀 케이스에서는 첫 번째 글자를 소문자로 적지 않음

캐멀 케이스 유형 1 : `PrintHello` → 파이썬에서 사용합니다.

캐멀 케이스 유형 2 : `printHello` → 파이썬에서 사용하지 않습니다.

`print`

`input`

`list`

`str`

`map`

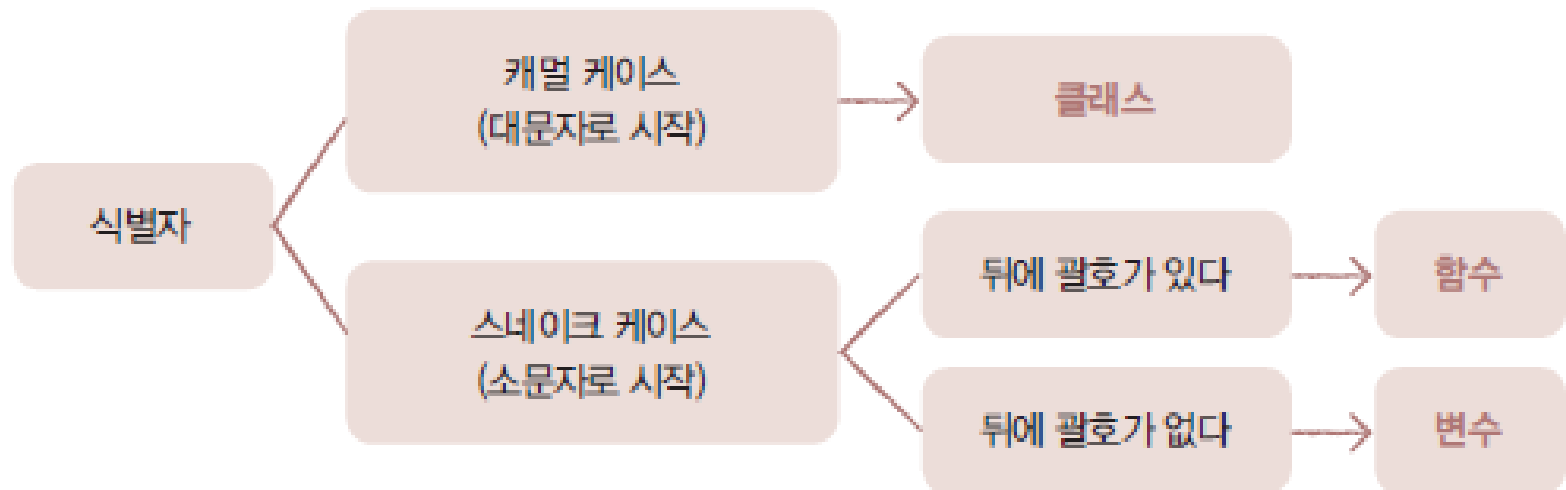
`filter`

`Animal`

`Customer`

식별자

- 캐멀 케이스로 작성되었으면 클래스
- 스네이크 케이스로 작성되어 있으면 함수 또는 변수
- 뒤에 괄호 붙으면 함수
- 뒤에 괄호 없으면 변수



식별자

- 식별자가 클래스인지, 변수인지, 함수인지 구분해 봅시다

```
1. print()  
2. list()  
3. soup.select()  
4. math.pi  
5. math.e  
6. class Animal:  
7. BeautifulSoup()
```

주석

- 주석 (comment)
 - 프로그램 진행에 영향 주지 않는 코드
 - 프로그램 설명 위해 사용
 - # 기호를 주석으로 처리하고자 하는 부분 앞에 붙임

```
>>> # 간단히 출력하는 예입니다.
>>> print("Hello! Python Programming...") # 문자열을 출력합니다.
Hello! Python Programming...
```

→ # 기호 뒷부분이 주석 처리됩니다.

연산자와 자료

- 연산자

- 스스로 값이 되는 것이 아닌 값과 값 사이에 무언가 기능 적용할 때 사용

```
>>> 1 + 1
2
>>> 10 - 10
0
```

- 리터럴 (literal)

- 자료 = 어떠한 값 자체

```
1
10
"Hello"
```

출력 : print()

- print() 함수
 - 출력 기능
 - 출력하고 싶은 것들을 괄호 안에 나열

```
print(출력1, 출력2, ...)
```

- 하나만 출력하기

```
>>> print("Hello! Python Programming...")
Hello! Python Programming...
>>> print(52)
52
>>> print(273)
273
```


출력 : print()

- 여러 개 출력하기

```
>>> print(52, 273, "Hello")
52 273 Hello
>>> print("안녕하세요", "저의", "이름은", "윤인성입니다!")
안녕하세요 저의 이름은 윤인성입니다!
```

- 줄바꿈하기

```
>>> print()
      → 빈 줄을 출력합니다.
>>>
```

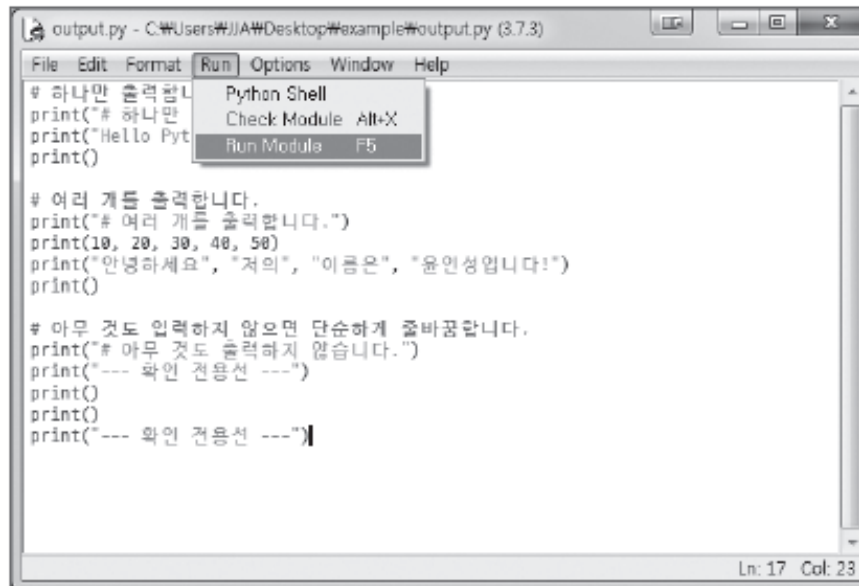
출력 : print()

- 예시 - 기본 출력

```
01  # 하나만 출력합니다.
02  print("# 하나만 출력합니다.")
03  print("Hello Python Programming...!")
04  print()
05
06  # 여러 개를 출력합니다.
07  print("# 여러 개를 출력합니다.")
08  print(10, 20, 30, 40, 50)
09  print("안녕하세요", "저의", "이름은", "윤인성입니다!")
10  print()
11
12  # 아무것도 입력하지 않으면 단순히 줄바꿈합니다.
13  print("# 아무것도 출력하지 않습니다.")
14  print("— 확인 전용선 —")
15  print()
16  print()
17  print("— 확인 전용선 —")
```

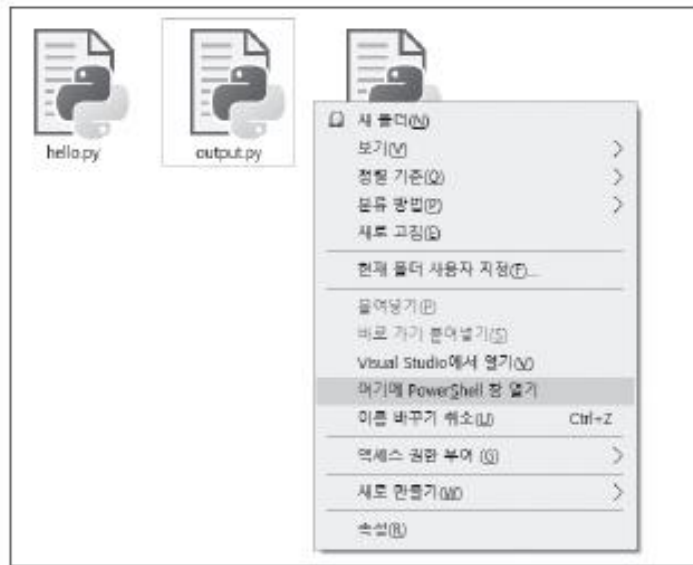
출력 : print()

- 파이썬 IDLE 에디터에서의 실행
 - [Run] – [Run Module] 선택



출력 : print()

- 비주얼 스튜디오 코드에서의 실행
 - 탐색기에서 파일 저장한 폴더로 이동하여 shift 누른 상태로 마우스 우클릭
 - [여기에 PowerShell 창 열기]



출력 : print()

- 명령 프롬프트 실행되면 python 명령어 사용하여 해당 파일 실행

```
> python output.py
```

```
# 하나만 출력합니다.
```

```
Hello Python Programming...!
```

```
# 여러 개를 출력합니다.
```

```
10 20 30 40 50
```

```
안녕하세요 저의 이름은 윤인성입니다!
```

```
# 아무것도 출력하지 않습니다.
```

```
— 확인 전용선 —
```

```
— 확인 전용선 —
```

키워드로 정리하는 핵심 포인트

- **표현식** : 값을 만들어내는 간단한 코드
- **키워드** : 의미가 부여된 특별한 단어로, 사용자가 지정하는 이름에 사용할 수 없음
- **식별자** : 프로그래밍 언어에서 이름 붙일 때 사용
- **주석** : 프로그램 설명 시 사용하며, 프로그램 자체에 영향 주지 않음
- **print()** : 파이썬의 가장 기본적인 출력방법으로, 괄호 안에 출력하고 싶은 것을 입력

1-2

자료형과 문자열

목차

- 시작하기 전에
- 자료형과 기본 자료형
- 문자열 만들기
- 문자열 연산자
- 문자열의 길이 구하기
- 키워드로 정리하는 핵심 포인트

시작하기 전에

[핵심 키워드] 자료형, 문자열, 이스케이프 문자, 문자열 연산자, `type()`, `len()`

[핵심 포인트] 프로그램이 처리할 수 있는 모든 것을 자료라 부른다. 자료란 무엇인지, 이를 처리하는 방법은 무엇인지, 그리고 가장 일반적으로 쓰이는 문자열 자료형은 어떤 것이 있는지 알아본다.

시작하기 전에

- 자료 (data)
 - 프로그램이 처리할 수 있는 모든 것
 - 프로그램은 자료를 처리하기 위한 모든 행위

자료형과 기본 자료형

- **자료형** (data type)

- 자료를 기능과 역할에 따라 구분한 것
- **문자열** (string) : 메일 제목, 메시지 내용 등
- **숫자** (number) : 물건의 가격, 학생의 성적 등
- **불** (boolean) : 친구의 로그인 상태 등

- **자료를 알아야 하는 이유**

- 자료를 자료형에 맞게 모으고 처리 과정을 거쳐 더 큰 자료형을 점차 만들어 나가면 프로그램이 완성됨
- 가장 기본 단위로서의 자료의 의미와 쓰임새를 아는 것은 매우 중요

자료형과 기본 자료형

- 자료형 (data type)
 - 자료의 형식
 - `type()` 함수로 확인

```
>>> print(type("안녕하세요"))  
<class 'str'>  
>>> print(type(273))  
<class 'int'>
```

- `str` : 문자열
- `int` : 정수

문자열 만들기

- 문자열 (string)
 - 따옴표로 둘러싸 입력하는, 글자가 나열된 것

"Hello" 'String' '안녕하세요' "Hello Python Programming"

```
# 하나만 출력합니다.
print("# 하나만 출력합니다.")
print("Hello Python Programming...!")
print()

# 여러 개를 출력합니다.
print("# 여러 개를 출력합니다.")
print(10, 20, 30, 40, 50)
print("안녕하세요", "저의", "이름은", "윤인성입니다!")
...
```

문자열

문자열

문자열

문자열 만들기

- 큰따옴표로 문자열 만들기

```
>>> print("안녕하세요")  
안녕하세요
```

- 작은따옴표로 문자열 만들기

```
>>> print('안녕하세요')  
안녕하세요
```

문자열 만들기

- 문자열 내부에 따옴표 넣기

"안녕하세요"라고 말했습니다

출력할 큰따옴표

```
>>> print("안녕하세요"라고 말했습니다)
```

문자열을 만들기 위해 사용한 큰따옴표

위 경우 오류 발생

- 파이썬 프로그래밍 언어는 자료와 자료를 단순 나열할 수 없음
- 구문 오류 (syntax error)

오류

SyntaxError: invalid syntax

문자열 만들기

- 작은따옴표로 문자열 만들어 큰따옴표 포함 문제 해결
 - 반대로도 가능

```
>>> print('"안녕하세요"라고 말했습니다')  
"안녕하세요"라고 말했습니다
```

```
>>> print("'배가 고픈다'라고 생각했습니다")  
'배가 고픈다'라고 생각했습니다
```


문자열 만들기

- 이스케이프 문자 (escape character)
 - 역슬래시 기호와 함께 조합해서 사용하는 특수한 문자
 - \“ : 큰따옴표를 의미
 - \‘ : 작은따옴표를 의미

```
>>> print("\안녕하세요\라고 말했습니다")
"안녕하세요"라고 말했습니다
>>> print('\배가 고픈다\'라고 생각했습니다')
'배가 고픈다'라고 생각했습니다
```

- \n : 줄바꿈 의미
- \t : 탭 의미

문자열 만들기

```
>>> print("안녕하세요\n안녕하세요")
```

```
안녕하세요
```

```
안녕하세요
```

```
>>> print("안녕하세요\t안녕하세요")
```

```
안녕하세요      안녕하세요
```

```
01 print("이름\n나이\n지역")
```

```
02 print("윤인성\n25\n강서구")
```

```
03 print("윤아린\n24\n강서구")
```

```
04 print("구름\n3\n강서구")
```

실행결과		
이름	나이	지역
윤인성	25	강서구
윤아린	24	강서구
구름	3	강서구

문자열 만들기

- \\: 역슬래시를 의미

```
>>> print("\\ \\ \\ \\")  
\\ \\ \\ \\
```

- 여러 줄 문자열 만들기

- \n 사용

```
>>> print("동해물과 백두산이 마르고 닳도록\n하느님이 보우하사 우리나라 만세\n무궁화 삼천리 화려강  
산 대한사람\n대한으로 길이 보전하세")  
동해물과 백두산이 마르고 닳도록  
하느님이 보우하사 우리나라 만세  
무궁화 삼천리 화려강산 대한사람  
대한으로 길이 보전하세
```

문자열 만들기

- 여러 줄 문자열 기능 활용: 큰따옴표 혹은 작은따옴표를 세 번 반복

```
>>> print("""동해물과 백두산이 마르고 닳도록  
하느님이 보우하사 우리나라 만세  
무궁화 삼천리 화려강산 대한사람  
대한으로 길이 보전하세""")  
동해물과 백두산이 마르고 닳도록  
하느님이 보우하사 우리나라 만세  
무궁화 삼천리 화려강산 대한사람  
대한으로 길이 보전하세
```

문자열 만들기

- 줄바꿈 없이 문자열 만들기
 - \ 기호 사용

```
>>> print("""  
동해물과 백두산이 마르고 닳도록  
하느님이 보우하사 우리나라 만세  
무궁화 삼천리 화려강산 대한사람  
대한으로 길이 보전하세  
""")
```

```
동해물과 백두산이 마르고 닳도록  
하느님이 보우하사 우리나라 만세  
무궁화 삼천리 화려강산 대한사람  
대한으로 길이 보전하세
```

→ 위 아래로 의도하지 않은 줄바꿈이 들어갑니다.

문자열 만들기

- 줄 뒤에 \ 붙여서 코드 쉽게 보기 위한 줄바꿈이며, 실질적 줄바꿈 아님을 나타냄

```
>>> print("""\
동해물과 백두산이 마르고 닳도록
하느님이 보우하사 우리나라 만세
무궁화 삼천리 화려강산 대한사람
대한으로 길이 보전하세\
""")
```

```
동해물과 백두산이 마르고 닳도록
하느님이 보우하사 우리나라 만세
무궁화 삼천리 화려강산 대한사람
대한으로 길이 보전하세
```

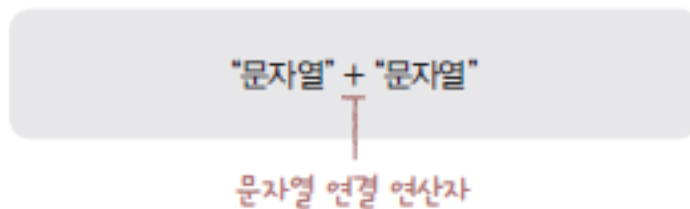
→ 줄을 바꿔 출력하지 않겠다고 선언합니다.

문자열 연산자

- 숫자에는 사칙연산 연산자를, 집합에는 여러 집합 연산자 적용 가능
 - 각 자료는 사용할 수 있는 연산자 정해져 있음



- 문자열 연결 연산자 : $+$



- 더하기와 같은 기호이나 다른 수행임에 주의


문자열 연산자

- 두 문자열 연결하여 새로운 문자열 만들어냄

```
>>> print("안녕" + "하세요")
안녕하세요
>>> print("안녕하세요" + "!")
안녕하세요!
```

- 문자열과 숫자 사이에는 사용할 수 없음

```
>>> print("안녕하세요" + 1)
```

 오류

TypeError: can only concatenate str (not "int") to str

- 문자열은 문자끼리, 숫자는 숫자끼리 연결
- 문자열과 숫자 연결하여 연산하려면 큰따옴표 붙여 문자열로 인식하게 함

문자열 연산자

- 문자열 반복 연산자 : *
- 문자열을 숫자와 * 연산자로 연결

```
>>> print("안녕하세요" * 3)  
안녕하세요안녕하세요안녕하세요
```

```
>>> print(3 * "안녕하세요")  
안녕하세요안녕하세요안녕하세요
```

문자열 연산자

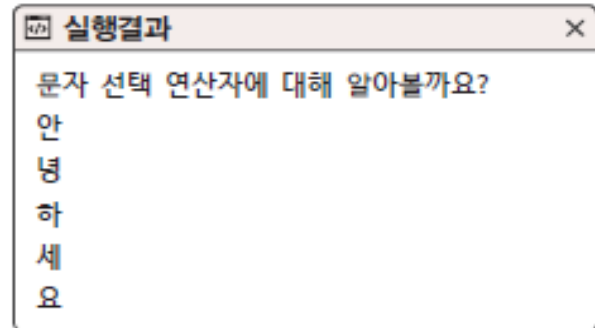
- 문자 선택 연산자 (인덱싱) : []
 - 문자열 내부의 문자 하나를 선택
 - 대괄호 안에 선택할 문자의 위치를 지정
- 인덱스 (index)
 - 제로 인덱스 (zero index) : 숫자를 0부터 셈
 - 원 인덱스 (one index) : 숫자를 1부터 셈
 - 파이썬은 제로 인덱스 유형 사용

안	녕	하	세	요
[0]	[1]	[2]	[3]	[4]

문자열 연산자

- 예시

```
01 print("문자 선택 연산자에 대해 알아보까요?")
02 print("안녕하세요"[0])
03 print("안녕하세요"[1])
04 print("안녕하세요"[2])
05 print("안녕하세요"[3])
06 print("안녕하세요"[4])
```



문자열 연산자

- 문자를 거꾸로 출력하려는 경우
 - 대괄호 안 숫자를 음수로 입력

안	녕	하	세	요
[-5]	[-4]	[-3]	[-2]	[-1]

```
01 print("문자를 뒤에서부터 선택해 볼까요?")
02 print("안녕하세요"[-1])
03 print("안녕하세요"[-2])
04 print("안녕하세요"[-3])
05 print("안녕하세요"[-4])
06 print("안녕하세요"[-5])
```

실행결과

문자를 뒤에서부터 선택해 볼까요?
요
세
하
녕
안

문자열 연산자

- 문자열 범위 선택 연산자 (슬라이싱) : [:]
 - 문자열의 특정 범위를 선택
 - 대괄호 안에 범위 구분 위치를 콜론으로 구분

```
>>> print("안녕하세요"[1:4])
```

녕하세

- 마지막 숫자 포함
- 마지막 숫자 포함하지 않음
 - 파이썬에서 적용

안	녕	하	세	요
[0]	[1]	[2]	[3]	[4]

문자열 연산자

- 예시

```
>>> print("안녕하세요"[0:2])
안녕
>>> print("안녕하세요"[1:3])
녕하
>>> print("안녕하세요"[2:4])
하세
```

- 대괄호 안에 넣는 숫자 둘 중 하나를 생략하는 경우
 - 뒤의 값 생략 : n번째부터 끝의 문자까지
 - 앞의 값 생략 : 0번째부터 뒤의 숫자 n번째 앞의 문자까지

[1:]
[:3]

```
>>> print("안녕하세요"[1:])
녕하세요
>>> print("안녕하세요"[:3])
안녕하
```

문자열 연산자

- **인덱싱** (indexing)
 - [] 기호 이용해 문자열의 특정 위치에 있는 문자 참조하는 것
- **슬라이싱** (slicing)
 - [:] 기호 이용해 문자열 일부를 추출하는 것
 - 문자열 선택 연산자로 슬라이스해도 원본은 변하지 않음에 주의

```
>>> hello = "안녕하세요" → ❶  
>>> print(hello[0:2]) → ❷  
안녕  
>>> hello → ❸  
'안녕하세요'
```

문자열 연산자

- **IndexError** (index out of range) 예외
 - 리스트/문자열 수를 넘는 요소/글자 선택할 경우 발생

```
>>> print("안녕하세요"[10])
```

❗ 오류

→ 파이썬 IDLE 에디터에서 실행했을 때 나타나는 내용으로 에디터마다 다르게 나타납니다.

Traceback (most recent call last):

File "<pyshell#2>", line 1, in <module>

print("안녕하세요"[10])

IndexError: string index out of range → IndexError 예외가 발생했어요.

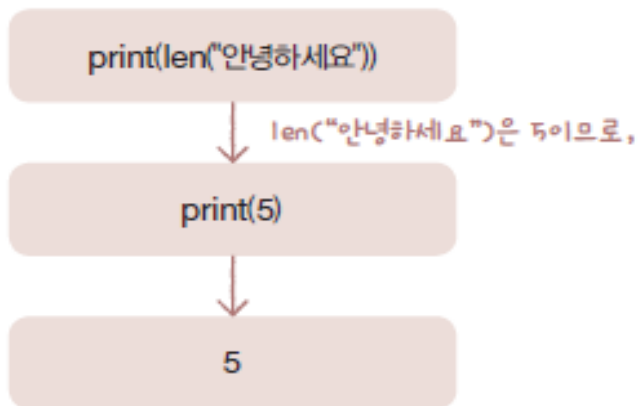
문자열의 길이 구하기

- len() 함수

- 문자열 길이 구할 때 사용
- 괄호 내부에 문자열 넣으면 문자열의 문자 개수 세어 줌

```
>>> print(len("안녕하세요"))  
5
```

- 중첩된 구조의 함수는 괄호 안쪽부터 먼저 실행



키워드로 정리하는 핵심 포인트

- **자료형** : 자료의 형식
- **문자열** : 문자의 나열. 큰따옴표 혹은 작은따옴표로 입력
- **이스케이프 문자** : 문자열 내부에서 특수한 기능 수행하는 문자열
- **문자열 연산자** : 문자열 연결 연산자 (+), 문자열 반복 연산자 (*), 문자열 선택 연산자 ([]), 문자열 범위 선택 연산자 ([:])
- **type()** : 자료형 확인하는 함수
- **len()** : 문자열 길이 구하는 함수

1-3

숫자

목차

- 시작하기 전에
- 숫자의 종류
- 숫자 연산자
- 연산자의 우선순위
- 키워드로 정리하는 핵심 포인트
- 확인문제

시작하기 전에

[핵심 키워드] 숫자 자료형, 숫자 연산자, 연산자, 우선순위

[핵심 포인트]

파이썬에서는 숫자를 소수점이 없는 숫자와 있는 숫자로 구분한다.

시작하기 전에

- 정수형

- 소수점이 없는 숫자
- 0, 1, 273, -52
- 정수 (integer)

- 실수형

- 소수점이 있는 숫자
- 0.0, 52.273, -1.2
- 실수 (floating point, 부동 소수점)

숫자의 종류

- 숫자를 만들기 위해서는 단순히 숫자 입력하면 됨

```
>>> print(273)
273
>>> print(52.273)
52.273
```

- type() 함수로 소수점 없는 숫자와 있는 숫자를 출력

```
>>> print(type(52))
<class 'int'>
>>> print(type(52.273))
<class 'float'>
```

- Int : 정수
- Float : 부동 소수점 (실수)
 - 일반적으로 프로그래밍 언어에서는 두 자료형을 구분해서 사용

숫자 연산자

- 사칙 연산자 : +, -, *, /
 - 덧셈, 뺄셈, 곱셈, 나눗셈

연산자	설명	구문	연산자	설명	구문
+	덧셈 연산자	숫자+숫자	*	곱셈 연산자	숫자*숫자
-	뺄셈 연산자	숫자-숫자	/	나눗셈 연산자	숫자/숫자

```
>>> print("5 + 7 =", 5 + 7)
5 + 7 = 12
>>> print("5 - 7 =", 5 - 7)
5 - 7 = -2
>>> print("5 * 7 =", 5 * 7)
5 * 7 = 35
>>> print("5 / 7 =", 5 / 7)
5 / 7 = 0.7142857142857143
```


숫자 연산자

- 정수 나누기 연산자: //

- 숫자를 나누고 소수점 이하 자릿수 삭제한 후 정수 부분만 남김

```
>>> print("3 / 2 =", 3 / 2)
3 / 2 = 1.5
>>> print("3 // 2 =", 3 // 2)
3 // 2 = 1
```

- 나머지 연산자 : %

- A를 B로 나누었을 때의 나머지를 구함

```
>>> print("5 % 2 =", 5 % 2)
5 % 2 = 1
```

숫자 연산자

- 제곱 연산자 : **
 - 숫자를 제곱함

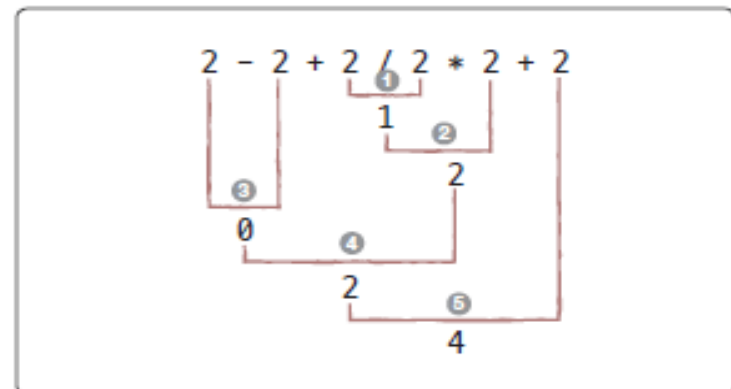
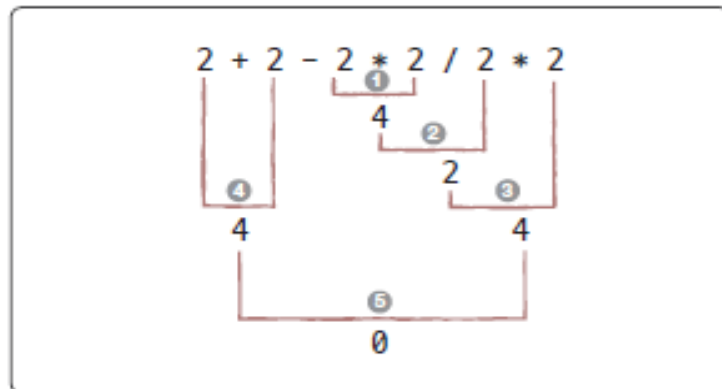
```
>>> print("2 ** 1 =", 2 ** 1)
2 ** 1 = 2
>>> print("2 ** 2 =", 2 ** 2)
2 ** 2 = 4
>>> print("2 ** 3 =", 2 ** 3)
2 ** 3 = 8
>>> print("2 ** 4 =", 2 ** 4)
2 ** 4 = 16
```

연산자의 우선순위

- 우선순위

- 파이썬의 수식은 연산자 간 우선순위에 따라 계산됨
 - 곱셈과 나눗셈이 덧셈과 뺄셈보다 우선

```
>>> print(2 + 2 - 2 * 2 / 2 * 2)
0.0
>>> print(2 - 2 + 2 / 2 * 2 + 2)
4.0
```



연산자의 우선순위

- 괄호 활용하여 우선순위 조정

$(5 + 3) * 2$

- 연산자 우선순위 확실한 경우에도 괄호로 감싸는 것이 좋음

$5 + (3 * 2)$

연산자의 우선순위

- TypeError 예외

- 서로 다른 자료를 연산할 경우

```
>>> string = "문자열"  
>>> number = 273  
>>> string + number
```

❗ 오류

```
Traceback (most recent call last):  
  File "<pyshell#3>", line 1, in <module>  
    string+number
```

`TypeError: can only concatenate str (not "int") to str` → TypeError 예외가 발생했어요.

키워드로 정리하는 핵심 포인트

- **숫자 자료형** : 소수점이 없는 정수형과 소수점이 있는 실수형 (부동 소수점) 이 있다.
- **숫자 연산자** : 사칙연산자와 // (정수 나누기 연산자), % (나누기 연산자), ** (제곱 연산자) 등이 있다.
- **연산자** : **우선순위**가 존재하는데, 곱하기와 나누기가 가장 우선이고 더하기와 빼기가 다음으로, 잘 모를 때는 괄호를 입력해 나타낸다.

1-4

변수와 입력

목차

- 시작하기 전에
- 변수 만들기/사용하기
- 복합 대입 연산자
- 사용자 입력 : `input()`
- 문자열을 숫자로 바꾸기
- 숫자를 문자열로 바꾸기
- 키워드로 정리하는 핵심 포인트

시작하기 전에

[핵심 키워드] 변수 선언, 변수 할당, 변수 참조, input(), int(), float(), str()

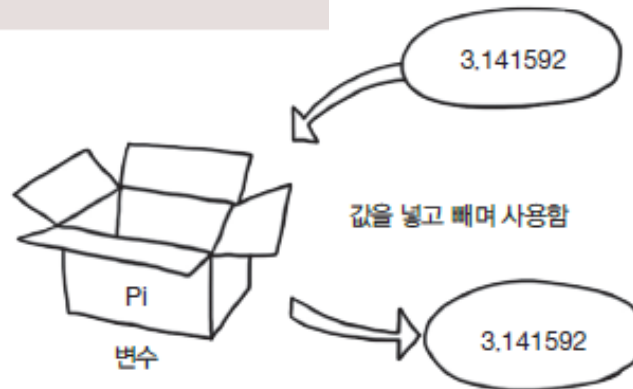
[핵심 포인트] 변수는 숫자뿐만이 아닌 모든 자료형을 의미하며, 파이썬에서 변수를 생성하는 것은 이를 사용하겠다고 선언하는 것이다. 변수에는 모든 자료형의 값을 저장할 수 있다.

시작하기 전에

- 변수

- 값을 저장할 때 사용하는 식별자
- 숫자뿐만 아니라 모든 자료형을 저장할 수 있음

```
>>> pi = 3.14159265  
>>> pi  
3.14159265
```



변수 만들기/사용하기

- 변수의 활용

- 변수를 선언하는 방법
 - 변수를 생성
- 변수에 값을 할당하는 방법
 - 변수에 값을 넣음
 - = 우변을 값을 좌변에 할당
- 변수를 참조하는 방법
 - 변수에서 값을 꺼냄
 - 변수 안에 있는 값을 사용

변수 = 값



값을 변수에 할당합니다.

변수 만들기/사용하기

- 변수를 참조

- 변수에 저장된 값을 출력

```
변수
```

- 변수에 저장된 값으로 연산

```
변수 + 변수
```

- 변수에 저장된 값을 출력

```
print(변수)
```

변수 만들기/사용하기

- 앞 예시에서 입력한 pi는 숫자 자료에 이름 붙인 것이기 때문에 숫자 연산 모두 수행할 수 있음

```
>>> pi = 3.14159265
>>> pi + 2
5.14159265
>>> pi - 2
1.1415926500000002
>>> pi * 2
6.2831853
>>> pi / 2
1.570796325
>>> pi % 2
1.1415926500000002
>>> pi * pi
9.869604378534024
```

변수 만들기/사용하기

- pi는 숫자 자료이므로 숫자와 문자열 연산은 불가능

```
pi + "문자열"
```

- 예시 - 원의 둘레와 넓이 구하기

```
01  # 변수 선언과 할당
02  pi = 3.14159265
03  r = 10
04
05  # 변수 참조
06  print("원주율 =", pi)
07  print("반지름 =", r)
08  print("원의 둘레 =", 2*pi*r)      # 원의 둘레
09  print("원의 넓이 =", pi*r*r)     # 원의 넓이
```

실행결과

```
원주율 = 3.14159265
반지름 = 10
원의 둘레 = 62.831853
원의 넓이 = 314.159265
```

복합 대입 연산자

- 복합 대입 연산자

- 기본 연산자와 = 연산자 함께 사용해 구성

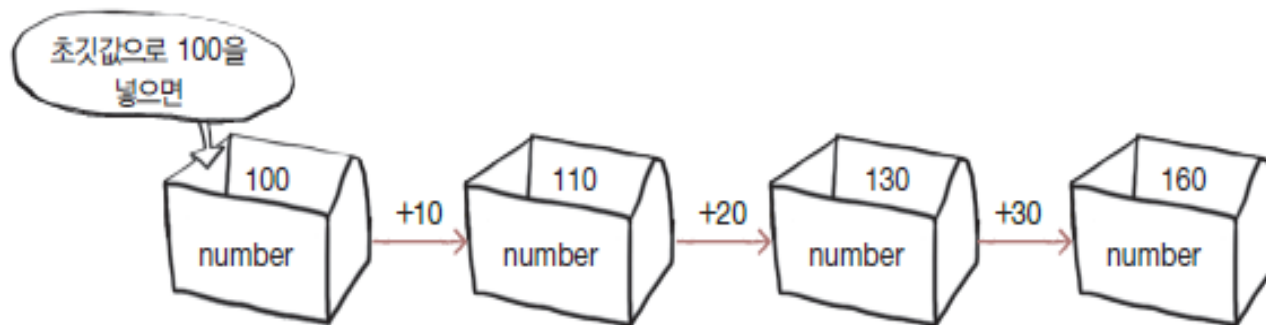
```
a += 10
```

연산자 이름	설명
+=	숫자 덧셈 후 대입
-=	숫자 뺄셈 후 대입
*=	숫자 곱셈 후 대입
/=	숫자 나눗셈 후 대입
%=	숫자의 나머지를 구한 후 대입
**=	숫자 제곱 후 대입

복합 대입 연산자

- 예시

```
>>> number = 100  
>>> number += 10  
>>> number += 20  
>>> number += 30  
>>> print("number:", number)  
number: 160
```



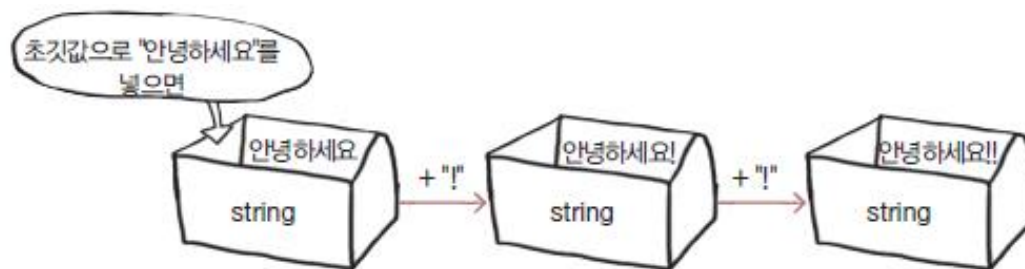
복합 대입 연산자

- 문자열 역시 복합 대입 연산자 사용 가능

연산자 이름	설명
<code>+=</code>	문자열 연결 후 대입
<code>*=</code>	문자열 반복 후 대입

- 예시

```
>>> string = "안녕하세요"
>>> string += "!"
>>> string += "!"
>>> print("string:", string)
string: 안녕하세요!!
```



사용자 입력 : input()

- input() 함수

- 명령 프롬프트에서 사용자로부터 데이터 입력받을 때 사용

- input() 함수로 사용자 입력받기

- 프롬프트 함수 : input 함수 괄호 안에 입력한 내용

```
>>> input("인사말을 입력하세요> ")
```

- 블록 (block) : 프로그램이 실행 중 잠시 멈추는 것

인사말을 입력하세요> | → 입력 대기를 알려주는 커서입니다. 커서는 프로그램에 따라 모양이 다를 수 있습니다.

- 명령 프롬프트에서 글자 입력 후 [Enter] 클릭

```
인사말을 입력하세요> 안녕하세요 
```

```
'안녕하세요'
```

사용자 입력 : input()

- input 함수의 결과로 산출 (리턴값)
 - 다른 변수에 대입하여 사용 가능

```
>>> string = input("인사말을 입력하세요> ")
```

```
인사말을 입력하세요> 안녕하세요 [Enter]
```

```
>>> print(string)
```

```
안녕하세요
```

사용자 입력 : input()

- input() 함수의 입력 자료형
 - type() 함수로 자료형 알아보기

```
>>> print(type(string))  
<class 'str'>
```

```
>>> number = input("숫자를 입력하세요> ")  
숫자를 입력하세요> 12345   
>>> print(number)  
12345
```

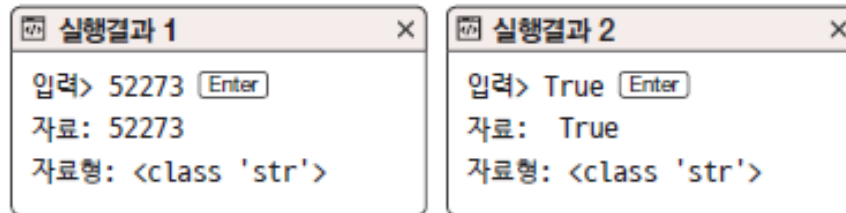
```
>>> print(type(number))  
<class 'str'>
```

- input() 함수는 사용자가 무엇을 입력해도 결과는 무조건 문자열 자료형

사용자 입력 : input()

- 예시 - 입력 자료형 확인하기

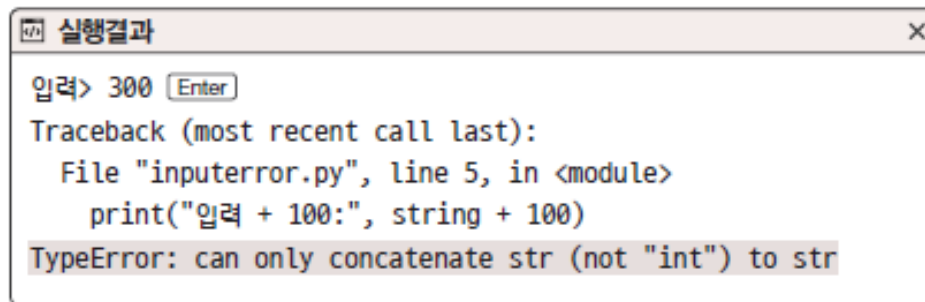
```
01  # 입력을 받습니다.  
02  string = input("입력> ")  
03  
04  # 출력합니다.  
05  print("자료:", string)  
06  print("자료형:", type(string))
```



사용자 입력 : input()

- 예시 - 입력받고 더하기

```
01  # 입력을 받습니다.  
02  string = input("입력> ")  
03  
04  # 출력합니다.  
05  print("입력 + 100:", string + 100)
```



The screenshot shows a window titled "실행결과" (Execution Result) with a close button. It displays the input "입력> 300" followed by an "Enter" button. Below this, a traceback is shown for a `TypeError`. The traceback indicates the error occurred in `inputerror.py` at line 5, within the `print` statement. The specific error message is `TypeError: can only concatenate str (not "int") to str`, which is highlighted in the original image.

```
실행결과  
입력> 300 Enter  
Traceback (most recent call last):  
  File "inputerror.py", line 5, in <module>  
    print("입력 + 100:", string + 100)  
TypeError: can only concatenate str (not "int") to str
```

문자열을 숫자로 바꾸기

- 캐스트 (cast)

- input() 함수의 입력 자료형은 항상 문자열이므로 입력받은 문자열을 숫자 연산에 활용하기 위해 숫자로 변환

- int() 함수

- 문자열을 int 자료형으로 변환.

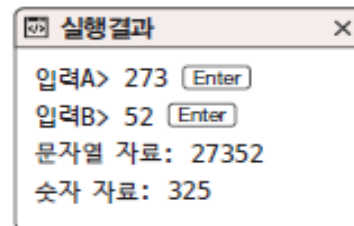
- float() 함수

- 문자열을 float 자료형으로 변환

문자열을 숫자로 바꾸기

- 예시 - int() 함수 활용하기

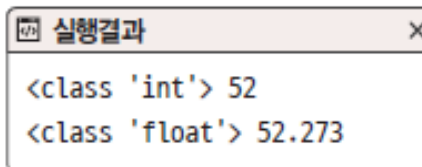
```
01 string_a = input("입력A> ")
02 int_a = int(string_a)
03
04 string_b = input("입력B> ")
05 int_b = int(string_b)
06
07 print("문자열 자료:", string_a + string_b)
08 print("숫자 자료:", int_a + int_b)
```



문자열을 숫자로 바꾸기

- 예시 – int() 함수와 float() 함수 활용하기

```
01 output_a = int("52")
02 output_b = float("52.273")
03
04 print(type(output_a), output_a)
05 print(type(output_b), output_b)
```

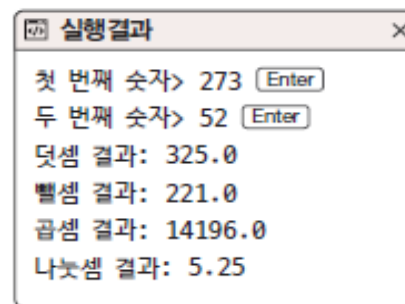


실행결과

```
<class 'int'> 52
<class 'float'> 52.273
```

- 예시 – int() 함수와 float 함수 조합하기

```
01 input_a = float(input("첫 번째 숫자> "))
02 input_b = float(input("두 번째 숫자> "))
03
04 print("덧셈 결과:", input_a + input_b)
05 print("뺄셈 결과:", input_a - input_b)
06 print("곱셈 결과:", input_a * input_b)
07 print("나눗셈 결과:", input_a / input_b)
```



실행결과


```
첫 번째 숫자> 273 Enter
두 번째 숫자> 52 Enter
덧셈 결과: 325.0
뺄셈 결과: 221.0
곱셈 결과: 14196.0
나눗셈 결과: 5.25
```

문자열을 숫자로 바꾸기

- ValueError 예외

- 변환할 수 없는 것을 변환하려 할 경우
- 숫자가 아닌 것을 숫자로 변환하려 할 경우

```
int("안녕하세요")  
float("안녕하세요")
```

 오류

```
Traceback (most recent call last):  
  File "intconvert.py", line 2, in <module>  
    int_a = int(string_a)  
ValueError: invalid literal for int() with base 10: '안녕하세요'
```

문자열을 숫자로 바꾸기

- 소수점이 있는 숫자 형식의 문자열을 int() 함수로 변환하려 할 때

```
int("52.273")
```

오류

```
Traceback (most recent call last):
```

```
File "intconvert.py", line 2, in <module>
```

```
    int_a = int(string_a)
```

```
ValueError: invalid literal for int() with base 10: '52.273'
```

숫자를 문자열로 바꾸기

- `str()` 함수
 - 숫자를 문자열로 변환

`str(다른 자료형)`

```
01 output_a = str(52)
02 output_b = str(52.273)
03 print(type(output_a), output_a)
04 print(type(output_b), output_b)
```

실행결과

```
<class 'str'> 52
<class 'str'> 52.273
```

키워드로 정리하는 핵심 포인트

- **변수 선언** : 변수를 생성하는 것을 의미
- **변수 할당** : 변수에 값을 넣는 것을 의미
- **변수 참조** : 변수에서 값을 꺼내는 것
- **input() 함수** : 명령 프롬프트에서 사용자로 부터 데이터 입력 받음
- **int() 함수** : 문자열을 int 자료형으로 변환
- **float 함수** : 문자열을 float 자료형으로 변환
- **str() 함수** : 숫자를 문자열로 변환

1-5

숫자와 문자열의 다양한 기능

목차

- 시작하기 전에
- 문자열의 `format()` 함수
- `format()` 함수의 다양한 기능
- 대소문자 바꾸기 : `upper()`와 `lower()`
- 문자열 양 옆의 공백 제거하기: `strip()`
- 문자열의 구성 파악하기 : `isOO()`
- 문자열 찾기: `find()`와 `rfind()`
- 문자열 자르기 : `split()`
- 키워드로 정리하는 핵심 포인트
- 확인문제

시작하기 전에

[핵심 키워드] format(), upper(), lower(), strip(), find(), in 연산자, split()

[핵심 포인트] 함수는 영어로 function, 즉 사람 또는 사물의 기능이라는 뜻을 가진 단어와 동음이의어다. 지금까지 살펴본 숫자나 문자열과 같은 자료도 컴퓨터에서는 하나의 사물처럼 취급되기에 내부적으로 여러 기능을 가지고 있다.

시작하기 전에

- 문자열 뒤에 마침표 입력해 보면 자동 완성 기능으로 다양한 자체 기능들이 제시됨

```
3 format_b = "파이썬 열공하여 첫 연봉 {}만 원 만들기".format(5000)
4 format_c = "{} {} {}".format(1, 2, 3)
5 format_d = "{} {} {}".format(1, 2, 3)
6 format_e = "{} {} {}".format(1, 2, 3)
7 format_f = "{} {} {}".format(1, 2, 3)
8
9 # 출력하기
10 print(format_a)
11 print(format_b)
12 print(format_c)
13 print(format_d)
14 print(format_e)
15 print(format_f)
```

capitalize
casefold
center
count
encode
endswith
expandtabs
find
format
format_map
index
isalnum

문자열의 format() 함수

- format() 함수로 숫자를 문자열로 변환

- 중괄호 포함한 문자열 뒤에 마침표 찍고 format() 함수 사용하되, 중괄호 개수와 format 함수 안 매개변수의 개수는 반드시 같아야 함
- 문자열의 중괄호 기호가 format() 함수 괄호 안의 매개변수로 차례로 대체되면서 숫자가 문자열이 됨

```
"{}".format(10)
```

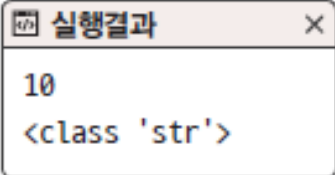
```
"{} {}".format(10, 20)
```

```
"{} {} {} {} {}".format(101, 202, 303, 404, 505)
```

문자열의 format() 함수

- 예시 – format() 함수로 숫자를 문자열로 변환하기

```
01  # format() 함수로 숫자를 문자열로 변환하기
02  string_a = "{}".format(10)
03
04  # 출력하기
05  print(string_a)
06  print(type(string_a))
```



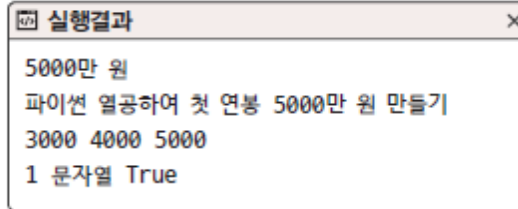
실행결과

```
10
<class 'str'>
```

문자열의 format() 함수

- 예시 – format() 함수의 다양한 형태

```
01  # format() 함수로 숫자를 문자열로 변환하기
02  format_a = "{}만 원".format(5000)
03  format_b = "파이썬 열공하여 첫 연봉 {}만 원 만들기 ".format(5000)
04  format_c = "{} {} {}".format(3000, 4000, 5000)
05  format_d = "{} {} {}".format(1, "문자열", True)
06
07  # 출력하기
08  print(format_a)
09  print(format_b)
10  print(format_c)
11  print(format_d)
```



실행결과

```
5000만 원
파이썬 열공하여 첫 연봉 5000만 원 만들기
3000 4000 5000
1 문자열 True
```

- format_a : 중괄호 옆에 다른 문자열 넣음
- format_b : 중괄호 앞뒤로 다른 문자열 넣음
- format_c : 매개변수 여러 개 넣음

문자열의 format() 함수

- IndexError 예외

- 중괄호 기호의 개수가 format() 함수의 매개변수 개수보다 많은 경우

```
>>> "{} {}".format(1, 2, 3, 4, 5)
'1 2'
>>> "{} {} {}".format(1, 2)
Traceback (most recent call last):
  File "<pyshell#1>", line 1, in <module>
    "{} {} {}".format(1, 2)
IndexError: tuple index out of range
```

format() 함수의 다양한 기능

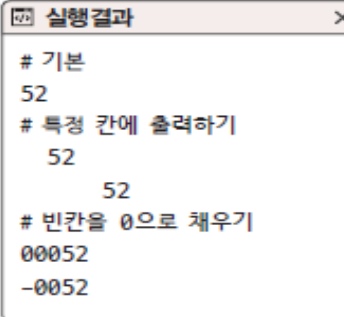
- 정수 출력의 다양한 형태
 - 예시 - 정수를 특정 칸에 출력하기

```
01  # 정수
02  output_a = "{:d}".format(52)
03
04  # 특정 칸에 출력하기
05  output_b = "{:5d}".format(52)      # 5칸
06  output_c = "{:10d}".format(52)     # 10칸
07
08  # 빈칸을 0으로 채우기
09  output_d = "{:05d}".format(52)     # 양수
10  output_e = "{:05d}".format(-52)    # 음수
11
12  print("# 기본")
13  print(output_a)
14  print("# 특정 칸에 출력하기")
15  print(output_b)
16  print(output_c)
17  print("# 빈칸을 0으로 채우기")
18  print(output_d)
19  print(output_e)
```

output_a : {:d}를 사용하여 int 자료형 정수 출력한
다는 것을 직접 지정

output_b, output_c : 특정 칸에 맞춰서 숫자를 출력
하는 형태

output_d, output_e : 빈칸을 0으로 채우는 형태

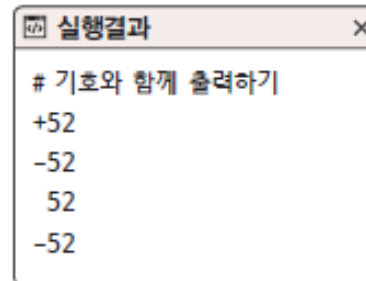


```
# 기본
52
# 특정 칸에 출력하기
52
52
# 빈칸을 0으로 채우기
00052
-0052
```

format() 함수의 다양한 기능

- 예시 - 기호 붙여 출력하기

```
01  # 기호와 함께 출력하기
02  output_f = "{:+d}".format(52)  # 양수
03  output_g = "{:+d}".format(-52) # 음수
04  output_h = "{: d}".format(52)  # 양수: 기호 부분 공백
05  output_i = "{: d}".format(-52) # 음수: 기호 부분 공백
06
07  print("# 기호와 함께 출력하기")
08  print(output_f)
09  print(output_g)
10  print(output_h)
11  print(output_i)
```



실행결과

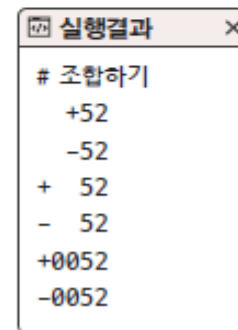
```
# 기호와 함께 출력하기
+52
-52
 52
-52
```

- {:+d} 앞에 + 기호 추가하면 양수의 경우 + 붙여줌
- {: d}처럼 앞에 공백두면 양수의 경우 기호 위치를 공백으로 비워줌

format() 함수의 다양한 기능

- 예시 - 조합

```
01  # 조합하기
02  output_h = "{:+5d}".format(52)      # 기호를 뒤로 밀기: 양수
03  output_i = "{:+5d}".format(-52)     # 기호를 뒤로 밀기: 음수
04  output_j = "{:=+5d}".format(52)     # 기호를 앞으로 밀기: 양수
05  output_k = "{:=+5d}".format(-52)    # 기호를 앞으로 밀기: 음수
06  output_l = "{:05d}".format(52)      # 0으로 채우기: 양수
07  output_m = "{:05d}".format(-52)    # 0으로 채우기: 음수
08
09  print("# 조합하기")
10  print(output_h)
11  print(output_i)
12  print(output_j)
13  print(output_k)
14  print(output_l)
15  print(output_m)
```

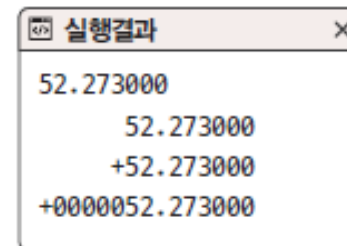


```
실행결과
# 조합하기
+52
-52
+ 52
- 52
+0052
-0052
```


format() 함수의 다양한 기능

- 부동 소수점 출력의 다양한 형태
 - 예시 - float 자료형 기본

```
01 output_a = "{:f}".format(52.273)
02 output_b = "{:15f}".format(52.273)    # 15칸 만들기
03 output_c = "{:+15f}".format(52.273)    # 15칸에 부호 추가하기
04 output_d = "{:+015f}".format(52.273)    # 15칸에 부호 추가하고 0으로 채우기
05
06 print(output_a)
07 print(output_b)
08 print(output_c)
09 print(output_d)
```



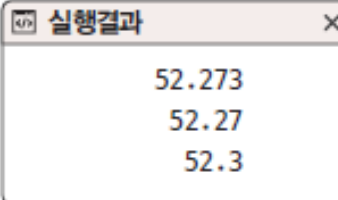
실행결과

```
52.273000
52.273000
+52.273000
+0000052.273000
```

format() 함수의 다양한 기능

- 예시 - 소수점 아래 자릿수 지정하기

```
01 output_a="{:15.3f}".format(52.273)
02 output_b="{:15.2f}".format(52.273)
03 output_c="{:15.1f}".format(52.273)
04
05 print(output_a)
06 print(output_b)
07 print(output_c)
```



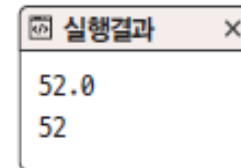
실행결과

52.273
52.27
52.3

format() 함수의 다양한 기능

- 의미 없는 소수점 제거하기
 - 예시 - `{:g}`

```
01 output_a = 52.0
02 output_b = "{:g}".format(output_a)
03 print(output_a)
04 print(output_b)
```



실행결과
52.0
52

대소문자 바꾸기 : upper()와 lower()

- upper() 함수
 - 문자의 알파벳을 대문자로 바꿈
- lower() 함수
 - 문자의 알파벳을 소문자로 바꿈

```
>>> a = "Hello Python Programming...!"  
>>> a.upper()  
'HELLO PYTHON PROGRAMMING...!'
```

```
>>> a.lower()  
'hello python programming...!'
```

문자열 양옆의 공백 제거하기: strip()

- strip() 함수
 - 문자열 양옆의 공백을 제거
- lstrip() 함수
 - 왼쪽의 공백을 제거
- rstrip() 함수
 - 오른쪽의 공백을 제거

문자열 양옆의 공백 제거하기: strip()

- 의도하지 않은 줄바꿈 등의 제거

```
>>> input_a = """
안녕하세요
문자열의 함수를 알아봅니다
"""
```

```
>>> print(input_a)
```

```

안녕하세요
문자열 함수를 알아봅니다
```

```
>>> print(input_a.strip())
```

```
안녕하세요
문자열 함수를 알아봅니다
```

문자열의 구성 파악하기 : isOO()

- 문자열이 소문자로만, 알파벳으로만, 혹은 숫자로만 구성되어 있는지 확인
 - `isalnum()`: 문자열이 알파벳 또는 숫자로만 구성되어 있는지 확인합니다.
 - `isalpha()`: 문자열이 알파벳으로만 구성되어 있는지 확인합니다.
 - `isidentifier()`: 문자열이 식별자로 사용할 수 있는 것인지 확인합니다.
 - `isdecimal()`: 문자열이 정수 형태인지 확인합니다.
 - `isdigit()`: 문자열이 숫자로 인식될 수 있는 것인지 확인합니다.
 - `isspace()`: 문자열이 공백으로만 구성되어 있는지 확인합니다.
 - `islower()`: 문자열이 소문자로만 구성되어 있는지 확인합니다.
 - `isupper()`: 문자열이 대문자로만 구성되어 있는지 확인합니다.

문자열의 구성 파악하기 : isOO()

- **불** (boolean)
 - 출력이 True 혹은 False로 나오는 것

```
>>> print("TrainA10".isalnum())
True
>>> print("10".isdigit())
True
```


문자열 찾기: find()와 rfind()

- find()
 - 왼쪽부터 찾아서 처음 등장하는 위치 찾을
- rfind()
 - 오른쪽부터 찾아서 처음 등장하는 위치 찾을

```
>>> output_a = "안녕안녕하세요".find("안녕")
>>> print(output_a)
0
```

```
>>> output_b = "안녕안녕하세요".rfind("안녕")
>>> print(output_b)
2
```

문자열과 in 연산자

- In 연산자

- 문자열 내부에 어떤 문자열이 있는지 확인할 때 사용
- 결과는 True(맞다), False(아니다)로 출력

```
>>> print("안녕" in "안녕하세요")  
True
```

```
>>> print("잘자" in "안녕하세요")  
False
```

문자열 자르기 : split()

- split() 함수
 - 문자열을 특정한 문자로 자름

```
>>> a = "10 20 30 40 50".split(" ")
>>> print(a)
['10', '20', '30', '40', '50']
```

- 실행 결과는 리스트 (list)로 출력

키워드로 정리하는 핵심 포인트

- **format() 함수** : 숫자와 문자열을 다양한 형태로 출력
- **upper() 및 lower() 함수** : 문자열의 알파벳을 대문자 혹은 소문자로 변경
- **strip() 함수** : 문자열 양옆의 공백 제거
- **find() 함수** : 문자열 내부에 특정 문자가 어디에 위치하는지 찾을 때 사용
- **in 연산자** : 문자열 내부에 어떤 문자열이 있는지 확인할 때 사용
- **split() 함수** : 문자열을 특정한 문자로 자를 때 사용

2-1

불 자료형과 if 조건문

목차

- 시작하기 전에
- 불 만들기 : 비교 연산자
- 불 연산하기 : 논리 연산자
- 논리 연산자의 활용
- if 조건문이란
- 날짜/시간 활용하기
- 컴퓨터의 조건
- 키워드로 정리하는 핵심 포인트

시작하기 전에

[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.

시작하기 전에

- Boolean

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

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

- 비교 연산자를 통해 만들 수 있음

연산자	설명	연산자	설명
==	같다	>	크다
!=	다르다	<=	작거나 같다
<	작다	>=	크거나 같다

불 만들기 : 비교 연산자

- 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
10 == 100	10과 100은 같다	거짓
10 != 100	10과 100은 다르다	참
10 < 100	10은 100보다 작다	참
10 > 100	10은 100보다 크다	거짓
10 <= 100	10은 100보다 작거나 같다	참
10 >= 100	10은 100보다 크거나 같다	거짓

불 만들기 : 비교 연산자

- 문자열에도 비교 연산자 적용 가능

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

불 연산하기 : 논리 연산자

- 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

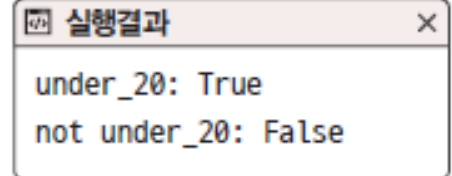
- 단항 연산자
- 참과 거짓 반대로 바꿈

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

불 연산하기 : 논리 연산자

- 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

불 연산하기 : 논리 연산자

- 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

불 연산하기 : 논리 연산자

- 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
```

논리 연산자의 활용

- and 연산자



- or 연산자



if 조건문이란

- 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.

if 조건문이란

- 예시

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

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

if 조건문이란

- 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

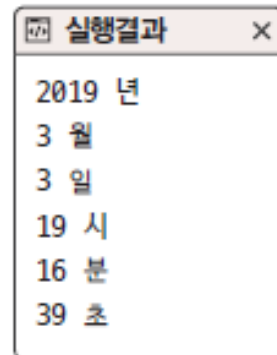
0입니다

날짜/시간 활용하기

- 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
```

날짜/시간 활용하기

- 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

날짜/시간 활용하기

- 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!

날짜/시간 활용하기

- 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:
```

날짜/시간 활용하기

```
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!

컴퓨터의 조건

- 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
```


컴퓨터의 조건

```
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")
```

실행결과 1

정수 입력> 52

짝수입니다

실행결과 2

정수 입력> 273

홀수입니다

컴퓨터의 조건

- 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")
```



컴퓨터의 조건

- 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")
```



키워드로 정리하는 핵심 포인트

- **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

2-2

불 자료형과 if 조건문

목차

- 시작하기 전에
- else 조건문의 활용
- elif 구문
- if 조건문을 효율적으로 사용하기
- False로 변환되는 값
- pass 키워드
- 키워드로 정리하는 핵심 포인트

시작하기 전에

[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.

시작하기 전에

- 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 조건문의 활용

- 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
```

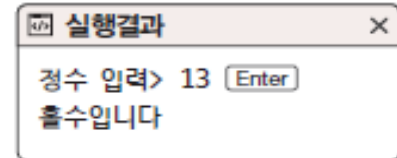
□□□□는 들여쓰기 4칸

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

else 조건문의 활용

- 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

- 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

elif 구문

- 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.")
```

실행결과

현재는 봄입니다

if 조건문을 효율적으로 사용하기

- 조건문의 활용

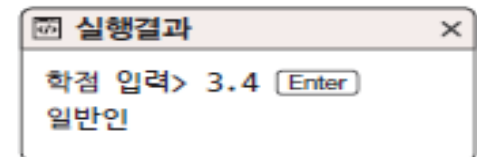
- 예시

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")
```

if 조건문을 효율적으로 사용하기

```
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

if 조건문을 효율적으로 사용하기

```
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")
```

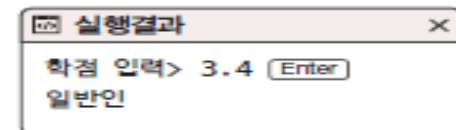
하위 값만 검사하고 상위 값은 검사를 생략

elif 4.2 <= score < 4.5:



elif 4.2 <= score:

조건 비교를 반으로 줄이고 코드 가독성 향상됨

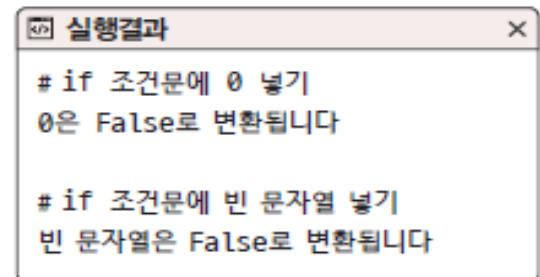


False로 변환되는 값

- 빈 컨테이너

- if 조건문의 매개변수에 불 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
```


pass 키워드

- 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.
```

pass 키워드

- IndentationError

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

- pass 키워드

- 아무것도 작성하지 않고 임시적으로 비워 둬

```
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
```

키워드로 정리하는 핵심 포인트

- **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

2-3

리스트와 반복문

목차

- 시작하기 전에
- 리스트 선언하고 요소에 접근하기
- 리스트 연산자: 연결(+), 반복(*), len()
- 리스트에 요소 추가하기: append, insert
- 리스트에 요소 제거하기
- 리스트 내부에 있는지 확인하기 : in/not in 연산자
- for 반복문
- for 반복문 : 리스트와 함께 사용하기
- 키워드로 정리하는 핵심 포인트
- 확인문제

시작하기 전에

[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.

시작하기 전에

- **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]
```

리스트 선언하고 요소에 접근하기

- **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 선언하고 요소에 접근하기

- 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

리스트 선언하고 요소에 접근하기

```
>>> 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 선언하고 요소에 접근하기

- 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]

리스트 선언하고 요소에 접근하기

- 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 선언하고 요소에 접근하기

- 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
```

리스트 선언하고 요소에 접근하기

- 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
```

리스트 연산자: 연결(+), 반복(*), len()

- 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)
```

리스트 연산자: 연결(+), 반복(*), len()

```
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
```


리스트 연산자: 연결(+), 반복(*), len()

- 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

리스트에 요소 추가하기: append, insert

- append() 함수
 - 리스트 뒤에 요소를 추가

```
리스트명.append(요소)
```

- insert() 함수
 - 리스트 중간에 요소를 추가

```
리스트명.insert(위치, 요소)
```

리스트에 요소 추가하기: append, insert

- 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]
```

리스트에 요소 추가하기: append, insert

- 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 위치
삽입할 값

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

↓
여번째 위치에 10을 추가합니다.

리스트에 요소 추가하기: append, insert

- `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]
```

리스트에 요소 추가하기: append, insert

- 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]
```

리스트에 요소 추가하기: append, insert

- 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

리스트에 요소 제거하기

- 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]
```


리스트에 요소 제거하기

- 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]

리스트에 요소 제거하기

- **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]
```

리스트에 요소 제거하기

- 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]
```

리스트에 요소 제거하기

- 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.
```

리스트 내부에 있는지 확인하기 : in/not in 연산자

- 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
```

리스트 내부에 있는지 확인하기 : in/not in 연산자

- 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
```

for 반복문

- 반복문
 - 컴퓨터에 반복 작업을 지시

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

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

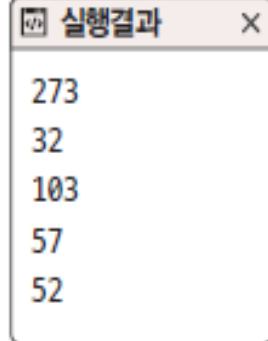
→ 반복에 사용할 수 있는 자료

for 반복문 : 리스트와 함께 사용하기

- 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
```


키워드로 정리하는 핵심 포인트

- **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

2-4

딕셔너리와 반복문

목차

- 시작하기 전에
- 딕셔너리 선언하기
- 딕셔너리의 요소에 접근하기
- 딕셔너리에 값 추가하기/제거하기
- 딕셔너리 내부에 키가 있는지 확인하기
- for 반복문 : 딕셔너리와 함께 사용하기
- 키워드로 정리하는 핵심 포인트
- 확인문제

시작하기 전에

[핵심 키워드] : 딕셔너리, 키, 값

[핵심 포인트]

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

시작하기 전에

- **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 선언하기

- 딕셔너리 선언
 - 중괄호로 선언하며 '키: 값' 형태를 쉼표로 연결해서 만들

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

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

딕셔너리의 요소에 접근하기

- 특정 키 값만 따로 출력하기
 - 딕셔너리 뒤에 대괄호 입력하고 그 내부에 키 입력

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

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

딕셔너리의 요소에 접근하기

- 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']
```


딕셔너리의 요소에 접근하기

구분	선언 형식	사용 예	틀린 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
```

딕셔너리의 요소에 접근하기

```
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: 당절임
ingredient: ['망고', '설탕', '메타중아황산나트륨', '치자황색소']
origin: 필리핀

name: 8D 건조 망고
```

딕셔너리의 요소에 접근하기

- 리스트 안의 특정 값 출력하려는 경우

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

딕셔너리의 요소에 접근하기

- 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
```

딕셔너리의 요소에 접근하기

- 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 추가하기/제거하기

- 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 대체

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

새로운 값으로 대체되었습니다.

딕셔너리에 값 추가하기/제거하기

- 딕셔너리 요소의 제거 : del 키워드

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

딕셔너리에 값 추가하기/제거하기

- 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'}

딕셔너리에 값 추가하기/제거하기

- 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: {}

딕셔너리에 값 추가하기/제거하기

- **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'
```

딕셔너리에 값 추가하기/제거하기

- 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'
```

딕셔너리 내부에 키가 있는지 확인하기

- 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: ")
```

딕셔너리 내부에 키가 있는지 확인하기

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

실행결과

> 접근하고자 하는 키: name Enter
70 건조 망고

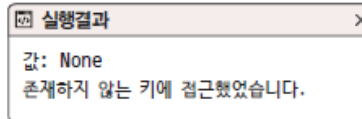
> 접근하고자 하는 키: ㅇ ㅂ ㅇ Enter
존재하지 않는 키에 접근하고 있습니다.

딕셔너리 내부에 키가 있는지 확인하기

- 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 반복문 : 딕셔너리와 함께 사용하기

- for 반복문과 딕셔너리의 조합

```
for 키 변수 in 딕셔너리:
```

```
    코드
```

for 반복문 : 딕셔너리와 함께 사용하기

```
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
```


키워드로 정리하는 핵심 포인트

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

2-5

딕셔너리와 반복문

목차

- 시작하기 전에
- 범위
- for 반복문: 범위와 함께 사용하기
- for 반복문 : 리스트와 범위 조합하기
- for 반복문: 반대로 반복하기
- while 반복문
- while 반복문 : for 반복문처럼 사용하기
- while 반복문: 상태를 기반으로 반복하기
- while 반복문 : 시간을 기반으로 반복하기
- while 반복문: break 키워드/continue 키워드
- 키워드로 정리하는 핵심 포인트

시작하기 전에

[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.

시작하기 전에

- 범위 (range)
 - 특정 횟수만큼 반복해서 돌리고 싶을 때 for 반복문과 조합하여 사용

범위

- 매개변수에 숫자를 한 개 넣는 방법
 - 0부터 A-1까지의 정수로 범위 만들

`range(A)` → A는 숫자

- 매개변수에 숫자를 두 개 넣는 방법
 - A부터 B-1까지의 정수로 범위 만들

`range(A, B)` → A와 B는 숫자

- 매개변수에 숫자를 세 개 넣는 방법
 - A부터 B-1까지의 정수로 범위 만들되 앞뒤의 숫자가 c만큼의 차이 가짐

`range(A, B, C)` → A, B, C는 숫자

범위

- 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]
```

범위

- 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

- 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

범위

- 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 반복문: 범위와 함께 사용하기

- 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
```

for 반복문 : 리스트와 범위 조합하기

- 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

for 반복문: 반대로 반복하기

- 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

for 반복문: 반대로 반복하기

- `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

- 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

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

while 반복문 : for 반복문처럼 사용하기

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

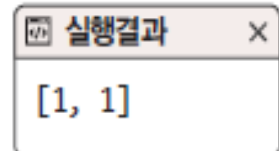
실행결과

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

while 반복문: 상태를 기반으로 반복하기

- 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]

while 반복문 : 시간을 기반으로 반복하기

- 예시 - 유닉스 타임 구하기
 - 시간 관련된 기능 가져오기

```
>>> import time
```

- 유닉스 타임

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

while 반복문 : 시간을 기반으로 반복하기

- 유닉스 타임과 while 반복문을 조합

```
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.

while 반복문: break 키워드/continue 키워드

- 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): ")
11      if input_text in ["y", "Y"]:
12          print("반복을 종료합니다.")
13          break
```

실행결과

0번째 반복문입니다
> 종료하시겠습니까?(y/n): n

1번째 반복문입니다
> 종료하시겠습니까?(y/n): n

2번째 반복문입니다
> 종료하시겠습니까?(y/n): n

3번째 반복문입니다
> 종료하시겠습니까?(y/n): n

4번째 반복문입니다
> 종료하시겠습니까?(y/n): y

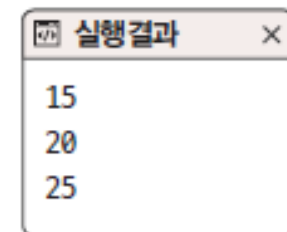
반복을 종료합니다.

while 반복문: break 키워드/continue 키워드

- 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)
```



while 반복문: break 키워드/continue 키워드

- 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
```

키워드로 정리하는 핵심 포인트

- **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

2-6

문자열, 리스트, 딕셔너리와 관련된 기본 함수

목차

- 시작하기 전에
- 리스트에 적용할 수 있는 기본 함수
- `reversed()` 함수로 리스트 뒤집기
- `enumerate()` 함수와 반복문 조합하기
- 딕셔너리의 `items()` 함수와 반복문 조합하기
- 리스트 내포
- 키워드로 정리하는 핵심 포인트
- 확인문제

시작하기 전에

[핵심 키워드] : enumerate(), items(), 리스트 내포

[핵심 포인트]

반복문과 관련된 파이썬만의 기능들에 대해 알아본다.

시작하기 전에

- 파이썬만의 고유한 기능들

리스트에 적용할 수 있는 기본 함수 : `min()`, `max()`, `sum()`

리스트 뒤집기 : `reversed()`

현재 인덱스가 몇 번째인지 확인하기 : `enumerate()`

딕셔너리로 쉽게 반복문 작성하기 : `item()`

리스트 안에 `for`문 사용하기 : 리스트 내포

리스트에 적용할 수 있는 기본 함수

- `min()`, `max()`, `sum()` 함수
 - 리스트를 매개변수로 넣어 사용하는 기본 함수들

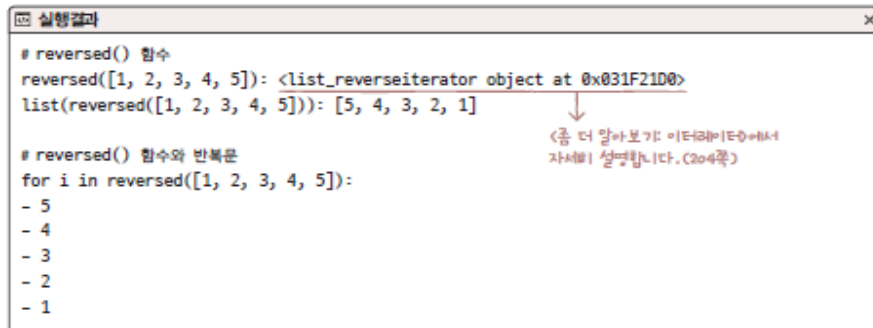
함수	설명
<code>min()</code>	리스트 내부에서 최솟값을 찾습니다.
<code>max()</code>	리스트 내부에서 최댓값을 찾습니다.
<code>sum()</code>	리스트 내부에서 값을 모두 더합니다.

```
>>> numbers = [103, 52, 273, 32, 77]
>>> min(numbers)           → 리스트 내부에서 최솟값을 찾습니다.
32
>>> max(numbers)           → 리스트 내부에서 최댓값을 찾습니다.
273
>>> sum(numbers)           → 리스트 내부에서 값을 모두 더합니다.
537
```

reversed() 함수로 리스트 뒤집기

- reversed() 함수
 - 리스트에서 요소 순서 뒤집기

```
01 # 리스트를 선언하고 뒤집습니다.
02 list_a = [1, 2, 3, 4, 5]
03 list_reversed = reversed(list_a)
04
05 # 출력합니다.
06 print("# reversed() 함수")
07 print("reversed([1, 2, 3, 4, 5]):", list_reversed)
08 print("list(reversed([1, 2, 3, 4, 5])):", list(list_reversed))
09 print()
10
11 # 반복문을 이용해 봅니다.
12 print("# reversed() 함수와 반복문")
13 print("for i in reversed([1, 2, 3, 4, 5]):")
14 for i in reversed(list_a):
15     print("-", i)
```



```
# reversed() 함수
reversed([1, 2, 3, 4, 5]): <list_reverseiterator object at 0x031F21D0>
list(reversed([1, 2, 3, 4, 5])): [5, 4, 3, 2, 1]

# reversed() 함수와 반복문
for i in reversed([1, 2, 3, 4, 5]):
- 5
- 4
- 3
- 2
- 1
```

↓
<좀 더 알아보기> 이터레이터(Iterator)에서 자세히 설명합니다. (204쪽)

reversed() 함수로 리스트 뒤집기

- reversed() 함수와 반복문 조합할 때는 함수 결과 여러 번 활용하지 않고 for 구문 내부에 reversed() 함수 곧바로 넣어서 사용
- 잘못된 예

```
temp = reversed([1, 2, 3, 4, 5, 6])

for i in temp:
    print("첫 번째 반복문: {}".format(i))

for i in temp:
    print("두 번째 반복문: {}".format(i))
```

```
첫 번째 반복문: 6
첫 번째 반복문: 5
첫 번째 반복문: 4
첫 번째 반복문: 3
첫 번째 반복문: 2
첫 번째 반복문: 1
```

reversed() 함수로 리스트 뒤집기

- 바른 예

```
numbers = [1, 2, 3, 4, 5, 6]
```

```
for i in reversed(numbers):  
    print("첫 번째 반복문: {}".format(i))
```

```
for i in reversed(numbers):  
    print("두 번째 반복문: {}".format(i))
```

→ 필요한 시점에 reversed() 함수를 사용합니다.

enumerate() 함수와 반복문 조합하기

- enumerate() 함수

- 리스트 요소 반복할 때 현재 인덱스가 몇 번째인지 확인
- 예시

```
example_list = ["요소A", "요소B", "요소C"]
```

0번째 요소는 요소A입니다.

1번째 요소는 요소B입니다.

2번째 요소는 요소C입니다.

enumerate() 함수와 반복문 조합하기

- 방법 1

```
example_list = ["요소A", "요소B", "요소C"]  
i = 0  
for item in example_list:  
    print("{}번째 요소는 {}".format(i, item))  
    i += 1
```

- 방법 2

```
example_list = ["요소A", "요소B", "요소C"]  
for i in range(len(example_list)):  
    print("{}번째 요소는 {}".format(i, example_list[i]))
```

enumerate() 함수와 반복문 조합하기

- 예시 – enumerate() 함수와 리스트

```
01  # 변수를 선언합니다.
02  example_list = ["요소A", "요소B", "요소C"]
03
04  # 그냥 출력합니다.
05  print("# 단순 출력")
06  print(example_list)
07  print()
08
09  # enumerate() 함수를 적용해 출력합니다.
10  print("# enumerate() 함수 적용 출력")
11  print(enumerate(example_list))
12  print()
13
14  # list() 함수로 강제 변환해 출력합니다.
15  print("# list() 함수로 강제 변환 출력")
16  print(list(enumerate(example_list)))
```

enumerate() 함수와 반복문 조합하기

```
17 print()
18
19 # for 반복문과 enumerate() 함수 조합해서 사용하기
20 print("# 반복문과 조합하기")
21 for i, value in enumerate(example_list):
22     print("{}번째 요소는 {}".format(i, value))
```

enumerate() 함수를 사용하면
반복 변수를 이런 형태로 넣을 수
있습니다.

실행결과

```
# 단순 출력
['요소A', '요소B', '요소C']

# enumerate() 함수 적용 출력
<enumerate object at 0x02A43CB0>

# list() 함수로 강제 변환 출력
[(0, '요소A'), (1, '요소B'), (2, '요소C')]

# 반복문과 조합하기
0번째 요소는 요소A입니다.
1번째 요소는 요소B입니다.
2번째 요소는 요소C입니다.
```

딕셔너리의 items() 함수와 반복문 조합하기

- 딕셔너리와 items() 함수 함께 사용하면 키와 값을 조합하여 쉽게 반복문 작성할 수 있음

```
01 # 변수를 선언합니다.
02 example_dictionary = {
03     "키A": "값A",
04     "키B": "값B",
05     "키C": "값C",
06 }
07
08 # 딕셔너리의 items() 함수 결과 출력하기
09 print("# 딕셔너리의 items() 함수")
10 print("items():", example_dictionary.items())
11 print()
12
13 # for 반복문과 items() 함수 조합해서 사용하기
14 print("# 딕셔너리의 items() 함수와 반복문 조합하기")
15
16 for key, element in example_dictionary.items():
17     print("dictionary[{}] = {}".format(key, element))
```

실행결과

```
# 딕셔너리의 items() 함수
items(): dict_items([('키A', '값A'), ('키B', '값B'), ('키C', '값C')])

# 딕셔너리의 items() 함수와 반복문 조합하기
dictionary[키A] = 값A
dictionary[키B] = 값B
dictionary[키C] = 값C
```

리스트 내포

- 반복문 사용하여 리스트 재조합하는 경우
 - 예시 - 반복문을 사용한 리스트 생성

```
01  # 변수를 선언합니다.  
02  array = []  
03  
04  # 반복문을 적용합니다.  
05  for i in range(0, 20, 2):  
06      array.append(i * i)  
07  # 출력합니다.  
08  print(array)
```

실행결과

[0, 4, 16, 36, 64, 100, 144, 196, 256, 324]

리스트 내포

- 예시 - 리스트 안에 for문 사용하기

```
01  # 리스트를 선언합니다.  
02  array = [i * i for i in range(0, 20, 2)]  
03           ↓  
           최종 결과를 앞에 작성합니다.  
04  # 출력합니다.  
05  print(array)
```

- 리스트 내포 (list comprehension)

리스트 이름 = [표현식 for 반복자 in 반복할 수 있는 것]

리스트 내포

- 예시 - 조건을 활용한 리스트 내포

리스트 이름 = [표현식 for 반복자 in 반복할 수 있는 것 if 조건문]

```
01  # 리스트를 선언합니다.  
02  array = ["사과", "자두", "초콜릿", "바나나", "체리"]  
03  output = [fruit for fruit in array if fruit != "초콜릿"]  
04  
05  # 출력합니다.  
06  print(output)
```

실행결과

['사과', '자두', '바나나', '체리']

키워드로 정리하는 핵심 포인트

- **enumerate() 함수** : 리스트를 매개변수로 넣을 경우 인덱스와 값을 쌍으로 사용해 반복문을 돌릴 수 있게 하는 함수
- **items() 함수** : 키와 쌍으로 사용해 반복문을 돌릴 수 있게 하는 딕셔너리 함수
- **리스트 내포** : 반복문과 조건문을 대괄호 안에 넣는 형태를 사용하여 리스트 생성하는 파이썬의 특수 구문. list comprehensions 기억할 것