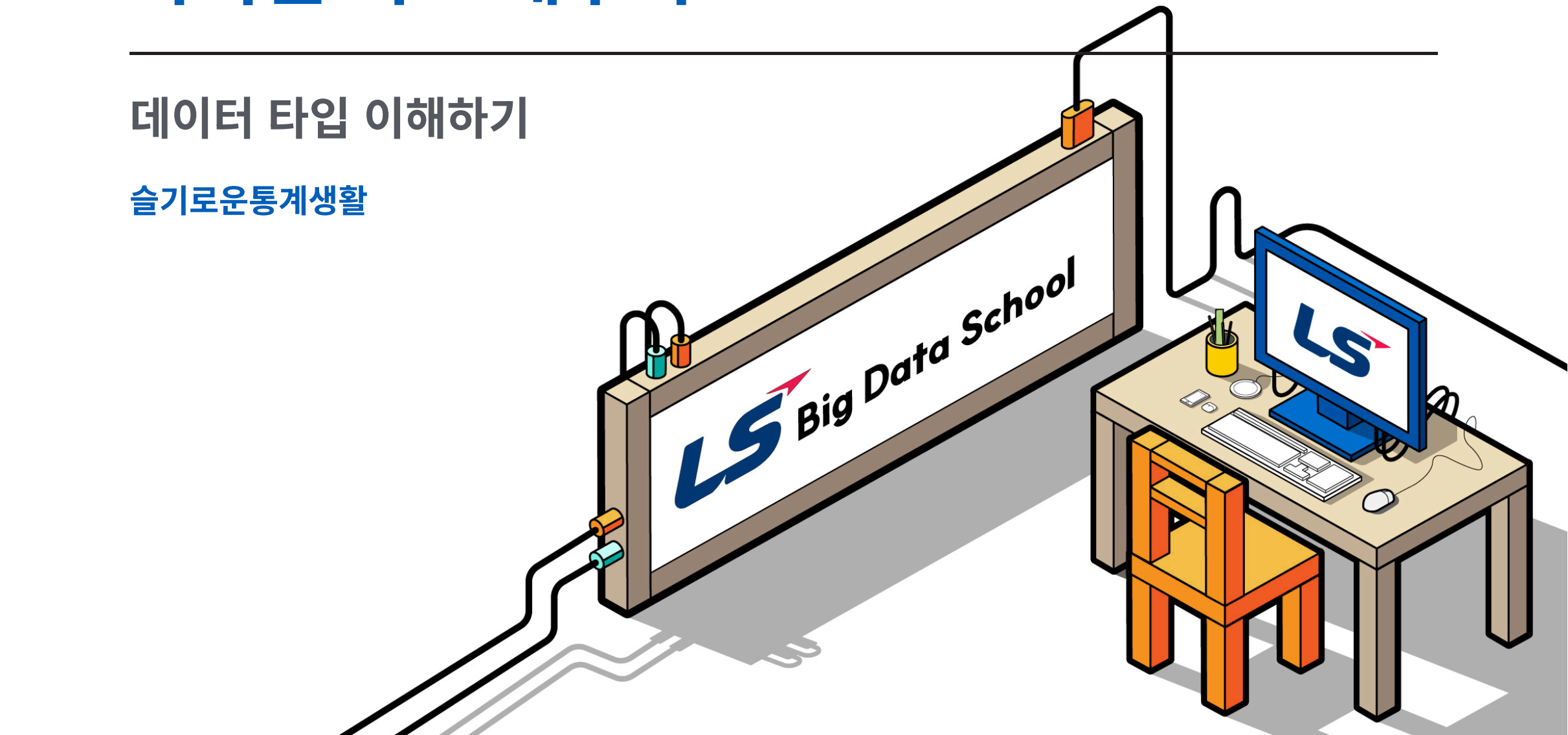
파이썬 기초 배우기

데이터 타입 이해하기

슬기로운통계생활



코스 훑어보기.

파이썬의 데이터 타입에 대해 학습합니다.

Python 데이터 타입 이해하기

파이썬은 데이터를 저장하고 처리하는 데 다양한 데이터 타입을 제공합니다. 이러한 데이터 타입을 이해하는 것은 파이썬 프로그래밍의 기초입니다.

파이썬 데이터 타입 큰 그림

파이썬은 다양한 데이터 타입을 지원하여, 프로그래머가 데이터를 효율적으로 관리하고 조작할 수 있게 합니다. 아래 표는 파이썬에서 사용되는 주요 데이터 타입들과 각각의 기본적인 특징을 정리합니다.

데이터 타입	설명	특징
숫자형	정수, 부동 소수점 수, 복소수 포함	변경 불가능(immutable)
문자열형	텍스트 데이터를 저장	변경 불가능(immutable)
리스트형	순서가 있는 항목의 컬렉션	변경 가능(mutable)
튜플형	순서가 있는 항목의 컬렉션	변경 불가능(immutable)
딕셔너리형	키와 값의 쌍을 저장	변경 가능(mutable)
집합형	중복 없는 요소의 컬렉션	변경 가능(mutable)
논리형	참(True) 또는 거짓(False) 값	변경 불가능(immutable)

Python의 숫자형 데이터 타입

파이썬에서 숫자형 데이터 타입은 수학적 계산에 필수적이며, 다양한 형태의 숫자를 표현할 수 있습니다. 숫자형 데이터 타입에는 `int` (정수), `float` (부동 소수점 숫자), 그리고 `complex` (복소수) 데이터 타입을 사용합니다.

각 데이터 타입에 대해 간단히 정리해보면 다음과 같습니다.

데이터 타입	설명	사용 예제
`int`	정수를 표현	`x = 10`
`float`	부동 소수점 수를 표현	`y = 3.14`
`complex`	복소수를 표현	`z = 1 + 2j`

숫자형 데이터 타입의 특징 및 사용 예제

정수형 (`int`)

정수형은 파이썬에서 가장 기본적인 숫자 타입으로, 양의 정수, 0, 음의 정수를 포함합니다.

```
x = 15
print(x, "는 ", type(x), "형식입니다.", sep='')
```

```
## 15는 <class 'int'>형식입니다.
```

숫자형 데이터 타입의 특징 및 사용 예제

부동 소수점형 (`float`)

부동 소수점형은 소수점을 포함하는 숫자를 표현합니다. 이 타입은 과학적 계산이나 정밀한 수치를 요구하는 계산에 주로 사용됩니다.

```
y = 3.14159
print(y, "는 ", type(y), "형식입니다.", sep='')
```

```
## 3.14159는 <class 'float'>형식입니다.
```

숫자형 데이터 타입의 특징 및 사용 예제

복소수형 (`complex`)

복소수형은 실수부와 허수부를 갖는 숫자를 표현합니다. 복소수는 일반적으로 공학이나 특정 과학 분야에서 사용됩니다.

```
z = 1 + 2j
print(z, "는 ", type(z), "형식입니다.", sep='')
```

```
## (1+2j)는 <class 'complex'>형식입니다.
```

복소수는 파이썬에서 `complex` 타입으로 표현되며, 실수부와 허수부(j로 표현 됨)를 갖습니다. 복소수 연산은 실수부와 허수부 각각에 대해 별도로 수행됩니다.

숫자형 데이터 타입의 특징 및 사용 예제

복소수형 (`complex`)

```
a = 1 + 2j  # 첫 번째 복소수
b = 3 + 4j  # 두 번째 복소수

# 복소수 덧셈
print("덧셈결과:", a + b)  # 출력: (4 + 6j)
```

```
## 덧셈결과: (4+6j)
```

```
# 복소수 뺄셈
print("뺄셈결과:", a - b)  # 출력: (-2 - 2j)
```

```
## 뺄셈결과: (-2-2j)
```

Python의 문자형 데이터 타입

문자형 데이터는 단일 문자 또는 문자열을 저장하며, 작은 따옴표(') 또는 큰 따옴표(")로 묶어서 표현합니다. 세 따옴표(''') 또는 (""")는 여러 줄에 걸친 문자열을 생성할 때 사용됩니다.

```
# 문자형 데이터 예제
a = "Hello, world!"
b = 'python programming'

# 여러 줄 문자열
ml_str = """This is
a multi-line
string"""

print(a, type(a))
```

```
## Hello, world! <class 'str'>
```

```
print(b, type(b))
```

```
## python programming <class 'str'>
```

```
print(ml_str, type(ml_str))
```

```
## This is
## a multi-line
## string <class 'str'>
```

문자열 연산

문자열은 `+` 연산자를 사용해 결합하고, `*` 연산자를 사용해 반복할 수 있습니다.

```
# 문자열 결합
greeting = "안녕" + " " + "파이썬!"
print("결합 된 문자열:", greeting)
```

```
## 결합 된 문자열: 안녕 파이썬!
```

```
# 문자열 반복
laugh = "하" * 3
print("반복 문자열:", laugh)
```

```
## 반복 문자열: 하하하
```

리스트 데이터 타입

리스트는 파이썬에서 가장 유연하고 많이 사용되는 데이터 구조 중 하나입니다. 리스트는 순서가 있고, 수정 가능한(mutable) 컬렉션으로서 다양한 타입의 요소들을 포함할 수 있습니다.

리스트형 (`list`) 데이터 타입 특징

리스트는 대괄호 `[]` 안에 쉼표로 구분된 요소들을 포함하여 생성됩니다. 리스트는 다양한 메소드를 통해 데이터를 관리할 수 있으며, 인덱싱과 슬라이싱을 통해 접근이 용이합니다. 또한, 리스트는 문자열, 숫자, 다른 리스트 등 어떠한 타입의 객체도 요소로 포함할 수 있습니다.

```
# 리스트 생성 예제
fruits = ['apple', 'banana', 'cherry']
numbers = [1, 2, 3, 4, 5]
mixed_list = [1, "Hello", [1, 2, 3]]
print("Fruits:", fruits)
```

```
## Fruits: ['apple', 'banana', 'cherry']
```

```
print("Numbers:", numbers)
```

```
## Numbers: [1, 2, 3, 4, 5]
```

```
print("Mixed List:", mixed_list)
```

```
## Mixed List: [1, 'Hello', [1, 2, 3]]
```

튜플형 데이터 타입

튜플(`tuple`)은 파이썬의 중요한 데이터 구조 중 하나로, 리스트와 유사하지만 한 번 생성된 후에는 수정할 수 없는(immutable) 특성을 가지고 있습니다. 이 변경 불가능성은 튜플을 데이터의 무결성을 유지해야 하는 상황에서 유용하게 만듭니다.

튜플은 소괄호 `()` 또는 괄호 없이 요소를 쉼표로 구분하여 생성할 수 있습니다. 튜플은 순서가 있으며, 중복된 요소를 포함할 수 있습니다.

튜플형 데이터 타입 특징과 생성 방법

튜플은 간단하게 요소들을 괄호 안에 나열하면 됩니다. 요소가 하나만 있는 튜플을 만들 때는 요소 뒤에 쉼표를 붙여야 합니다.

```
# 튜플 생성 예제
a = (10, 20, 30) # a = 10, 20, 30 과 동일
b = (42,)

print("좌표:", a)
```

```
## 좌표: (10, 20, 30)
```

```
print("단원소 튜플:", b)
```

```
## 단원소 튜플: (42,)
```

튜플은 리스트보다 더 적은 공간을 사용하고, 생성 후 변경할 수 없기 때문에 실행 시간이 빠릅니다. 또한, 튜플은 딕셔너리의 키와 같은 변경 불가능한 객체가 필요한 곳에서 사용될 수 있습니다.

튜플의 인덱싱과 슬라이싱

튜플도 리스트처럼 인덱싱과 슬라이싱이 가능합니다. 이를 통해 튜플의 특정 부분에 접근하거나 튜플의 일부를 추출할 수 있습니다.

```
# 인덱싱
print("첫번째 좌표:", a[0])
```

```
## 첫번째 좌표: 10
```

```
# 슬라이싱
print("마지막 두개 좌표:", a[1:])
```

```
## 마지막 두개 좌표: (20, 30)
```

튜플과 함수

튜플은 함수에서 여러 값을 반환할 때 자주 사용됩니다. 이를 통해 한 번의 함수 호출로 여러 결과를 동시에 반환할 수 있습니다.

```
def min_max(numbers):
    return min(numbers), max(numbers)

result = min_max([1, 2, 3, 4, 5])
result = min_max((1, 2, 3, 4, 5))
print("Minimum and maximum:", result)
```

```
## Minimum and maximum: (1, 5)
```

위 함수를 돌려보면 마지막 결과가 튜플로 반환 되었음을 알 수 있습니다.

튜플은 그 구조상 변경이 불가능하여 데이터의 안정성을 보장할 수 있습니다. 특히, 함수로부터 여러 값을 안전하게 반환하거나, 다양한 데이터 아이템을 묶어서 관리할 때 유용하게 사용됩니다.

딕셔너리(Dictionary)형 데이터 타입

딕셔너리는 파이썬의 중요한 데이터 구조 중 하나로, 키(`key`)와 값(`value`)의 쌍으로 데이터를 저장합니다. 딕셔너리는 데이터에 빠르게 접근할 수 있도록 해주며, 데이터의 추가, 삭제, 수정이 자유롭습니다.

딕셔너리형 (`dict`) 데이터 타입 개요

딕셔너리는 중괄호 `{}`를 사용하여 생성되며, 각 항목은 `키: 값` 형태로 표현됩니다. 키는 변경 불가능한(immutable) 타입이어야 하며, 보통 문자열이나 숫자, 튜플이 사용됩니다.

딕셔너리 생성

딕셔너리를 생성하는 가장 간단한 방법은 중괄호 `{}` 안에 `키: 값` 쌍을 나열하는 것입니다.

```
# 딕셔너리 생성 예제
person = {
    'name': 'John',
    'age': 30,
    'city': 'New York'
}
print("Person:", person)
```

```
## Person: {'name': 'John', 'age': 30, 'city': 'New York'}
```

딕셔너리는 효율적인 데이터 검색, 수정 및 관리를 위한 다양한 부가기능(메서드)들을 제공합니다.
주요 메서드로는 `get()`, `update()`, `keys()`, `values()`, `items()` 등이 있습니다.

- `get()`을 사용한 값 빼내오기

```
print(person.get('name'))
```

```
## John
```

- `keys()`을 사용한 키 빼내오기

```
print(person.keys())
```

```
## dict_keys(['name', 'age', 'city'])
```

- `values()`을 사용한 값 빼내오기

```
print(person.values())
```

```
## dict_values(['John', 30, 'New York'])
```

집합형 데이터 타입

집합(set)은 파이썬의 데이터 구조 중 하나로, 순서가 없고 중복된 요소를 허용하지 않는 컬렉션입니다. 집합은 수학적인 집합 연산을 지원하며, 주로 중복을 제거하거나 항목의 존재 여부를 빠르게 검사할 때 사용됩니다.

집합형 (`set`) 데이터 타입 개요

집합은 중괄호 `{}`를 사용하여 생성하거나 `set()` 생성자를 통해 생성할 수 있습니다. 집합은 변경 가능(mutable)하며, 한 번에 하나의 유니크한 요소만 저장할 수 있습니다. 또한, 집합은 중복된 값을 자동으로 필터링하며, 입력된 순서를 유지하지 않습니다.

집합 생성

집합을 생성하는 가장 간단한 방법은 중괄호 `{}` 안에 값을 나열하는 것입니다.

```
# 집합 생성 예제
fruits = {'apple', 'banana', 'cherry', 'apple'}
print("Fruits set:", fruits)  # 중복 'apple'은 제거됨
```

```
## Fruits set: {'cherry', 'banana', 'apple'}
```

```
# 빈 집합 생성
empty_set = set()
print("Empty set:", empty_set)
```

```
## Empty set: set()
```

집합에서 사용 가능한 부가기능(메서드) 정리

집합은 요소의 추가, 삭제 및 집합 간 연산을 지원합니다. `add()`, `remove()`, `update()`, `union()`, `intersection()` 등의 메서드를 제공합니다. 다음은 집합에서 사용가능 한 메서드 정리표입니다.

메서드	설명
`add(element)`	집합에 요소를 추가합니다.
`remove(element)`	집합에서 요소를 제거합니다. 요소가 집합에 없으면 `KeyError` 발생.
`discard(element)`	집합에서 요소를 제거합니다. 요소가 집합에 없어도 에러가 발생하지 않습니다.
`pop()`	집합에서 임의의 요소를 제거하고 해당 요소를 반환합니다. 집합이 비어 있으면 `KeyError` 발생.
`clear()`	집합의 모든 요소를 제거합니다.
`update(other)`	다른 집합 또는 반복 가능한 객체의 요소를 현재 집합에 추가합니다.
`union(other)`	현재 집합과 다른 집합의 합집합을 새 집합으로 반환합니다.

집합에서 사용 가능한 부가기능(메서드) 정리

메서드	설명
`intersection(other)`	현재 집합과 다른 집합의 교집합을 새 집합으로 반환합니다.
`difference(other)`	현재 집합과 다른 집합의 차집합을 새 집합으로 반환합니다.
`symmetric_difference(other)`	현재 집합과 다른 집합의 대칭 차집합을 새 집합으로 반환합니다.
`isdisjoint(other)`	두 집합이 겹치는 요소가 없으면 True를 반환합니다.
`issubset(other)`	현재 집합이 다른 집합의 부분집합이면 True를 반환합니다.
`issuperset(other)`	현재 집합이 다른 집합을 포함하면 True를 반환합니다.

집합형 데이터 타입 메서드 예제

실제 코드를 통하여 확인해봅시다.

```
# 요소 추가
fruits.add('orange')
print("After adding orange:", fruits)
```

```
## After adding orange: {'orange', 'cherry'
```

```
# 요소 삭제
fruits.remove('banana')
print("After removing banana:", fruits)
```

```
## After removing banana: {'orange', 'cherr
```

```
# 집합 간 연산
other_fruits = {'berry', 'cherry'}
union_fruits = fruits.union(other_fruits)
intersection_fruits = fruits.intersection(c
print("Union of fruits:", union_fruits)
```

```
## Union of fruits: {'cherry', 'orange', 'a
```

```
print("Intersection of fruits:", intersecti
```

```
## Intersection of fruits: {'cherry'}
```

논리형 데이터 타입

논리 연산자와 논리형 데이터 타입

논리형 데이터 타입은 앞에서 배웠던 논리 연산자 `and`, `or`, `not`과 같이 사용되기도 합니다.

```
# 논리 AND 연산
print("True and False:", True and False)  # False
```

```
## True and False: False
```

```
# 논리 OR 연산
print("True or False:", True or False)  # True
```

```
## True or False: True
```

```
# 논리 NOT 연산
print("Not True:", not True)  # False
```

```
## Not True: False
```

조건문에서의 사용

논리형은 주로 조건문에서 사용되어 프로그램의 분기를 결정합니다. 앞으로 배울 조건문 중 가장 기본이 되는 if 문을 살펴봅시다.

```
a=3
if (a == 2):
    print("a는 2와 같습니다.")
else:
    print("a는 2와 같지 않습니다.")
```

```
## a는 2와 같지 않습니다.
```

위의 조건문에서 논리형 데이터 타입은 바로 조건 체크에 사용되었습니다.

```
a == 2
```

```
## False
```

데이터 타입 변환하기

파이썬에서는 다양한 데이터 타입 간에 변환을 수행할 수 있습니다. 이러한 변환은 프로그램에서 데이터를 다루는 과정에서 유연성을 제공합니다. 숫자형, 문자열형, 리스트형, 튜플형, 딕셔너리형, 집합형, 논리형 데이터 타입 간의 변환 방법을 알아봅시다.

데이터 타입 변환 요약 표

변환 종류	사용 함수	설명
숫자형 → 문자열형	`str()`	숫자를 문자열로 변환
문자열형 → 숫자형	`int()`, `float()`	문자열을 정수 또는 부동 소수점 수로 변환
리스트 ↔ 튜플	`list()`, `tuple()`	리스트를 튜플로, 튜플을 리스트로 변환
리스트, 튜플 → 집합	`set()`	리스트나 튜플을 집합으로 변환
집합 → 리스트, 튜플	`list()`, `tuple()`	집합을 리스트나 튜플로 변환
집합 → 딕셔너리	(Not directly possible)	직접적인 함수 없음, 키 또는 값으로 구성 필요
딕셔너리 → 집합	`set()`	딕셔너리의 키 또는 값으로 집합을 생성
논리형 ↔ 숫자형	`bool()`, `int()`	논리값을 숫자로, 숫자를 논리값으로 변환
논리형 ↔ 문자열형	`str()`, `bool()`	논리값을 문자열로, 문자열을 논리값으로 변환

각 변환에 대한 세부 설명은 다음과 같습니다.

숫자형과 문자열형

- **숫자형 → 문자열형**: `str()` 함수를 사용하여 숫자를 문자열로 변환할 수 있습니다.
- **문자열형 → 숫자형**: `int()` 또는 `float()` 함수를 사용하여 문자열을 숫자로 변환할 수 있습니다. 단, 문자열이 숫자 형태를 띄고 있어야 합니다.

```
# 숫자형을 문자열형으로 변환
num = 123
str_num = str(num)
print("문자열:", str_num, type(str_num))
```

```
## 문자열: 123 <class 'str'>
```

```
# 문자열형을 숫자형(실수)으로 변환
num_again = float(str_num)
print("숫자형:", num_again, type(num_again))
```

```
## 숫자형: 123.0 <class 'float'>
```

리스트, 튜플, 집합

- **리스트 ↔ 튜플**: `list()` 함수로 튜플을 리스트로 변환하거나, `tuple()` 함수로 리스트를 튜플로 변환할 수 있습니다.
- **리스트 ↔ 집합, 튜플 ↔ 집합**: `set()` 함수를 사용하여 리스트나 튜플을 집합으로 변환할 수 있습니다. 집합에서 리스트나 튜플로의 변환도 각각 `list()` 또는 `tuple()` 함수를 사용합니다.

```
# 리스트와 튜플 변환
lst = [1, 2, 3]
print("리스트:", lst)
```

```
## 리스트: [1, 2, 3]
```

```
tup = tuple(lst)
print("튜플:", tup)
```

```
## 튜플: (1, 2, 3)
```

집합과 딕셔너리

- **집합 → 딕셔너리**: 집합은 직접적으로 딕셔너리로 변환할 수 없습니다. 집합의 각 요소를 딕셔너리의 키 또는 값으로 사용하여 새로운 딕셔너리를 생성할 수는 있습니다.
- **딕셔너리 → 집합**: 딕셔너리의 키 또는 값만을 추출하여 집합으로 변환할 수 있습니다. 예를 들어, `set(dictionary.keys())` 또는 `set(dictionary.values())`를 사용합니다.

집합과 딕셔너리 변환 예제는 다음과 같습니다. 직접적인 변환은 불가능하나 키 또는 값으로 구성할 수 있습니다.

```
set_example = {'a', 'b', 'c'}
# 딕셔너리로 변환 시, 일반적으로 집합 요소를 키 또는 값으로 사용
dict_from_set = {key: True for key in set_example}
print("Dictionary from set:", dict_from_set)
```

```
## Dictionary from set: {'b': True, 'a': True, 'c': True}
```

논리형과 다른 타입

- **논리형 ↔ 숫자형**: `bool()` 함수를 사용하여 숫자를 논리형으로 변환할 수 있으며 (`0`은 `False`, 그 외는 `True`), `int()` 함수로 논리형을 숫자로 변환할 수 있습니다 (`True`는 `1`, `False`는 `0`).
- **논리형 ↔ 문자열형**: `str()` 함수로 논리형을 문자열로 변환할 수 있고 (`True`는 `"True"`, `False`는 `"False"`), 문자열 `"True"`나 `"False"`는 `bool()` 함수로 논리형으로 변환할 수 있습니다.

논리형과 다른 타입 예제

```
# 논리형과 숫자형 변환 예제
# 숫자를 논리형으로 변환
zero = 0
non_zero = 7
bool_from_zero = bool(zero)           # Fals
bool_from_non_zero = bool(non_zero)   # True
```

```
print("0를 논리형으로 바꾸면:", bool_from_zero)
```

```
## 0를 논리형으로 바꾸면: False
```

```
print("7를 논리형으로 바꾸면:", bool_from_non_ze
```

```
## 7를 논리형으로 바꾸면: True
```

논리형을 숫자로 변환 예제

```
# 논리형을 숫자로 변환
true_bool = True
false_bool = False
int_from_true = int(true_bool)    # 1
int_from_false = int(false_bool)  # 0
print("True는 숫자로:", int_from_true)
```

```
## True는 숫자로: 1
```

```
print("False는 숫자로:", int_from_false)
```

```
## False는 숫자로: 0
```

논리형과 문자열형 변환

```
# 논리형과 문자열형 변환 예제
# 논리형을 문자열로 변환
str_from_true = str(true_bool)    # "True"
str_from_false = str(false_bool)  # "False"
print("True는 문자열로:", str_from_true)
```

```
## True는 문자열로: True
```

```
print("False는 문자열로:", str_from_false)
```

```
## False는 문자열로: False
```

문자열을 논리형으로 변환

```
# 문자열을 논리형으로 변환
str_true = "True"
str_false = "False"
bool_from_str_true = bool(str_true)    # True
bool_from_str_false = bool(str_false)  # True, 비어있지 않으면 무조건 참
print("'True'는 논리형으로 바꾸면:", bool_from_str_true)
```

```
## 'True'는 논리형으로 바꾸면: True
```

```
print("'False'는 논리형으로 바꾸면:", bool_from_str_false)
```

```
## 'False'는 논리형으로 바꾸면: True
```

문제 1: 숫자형 데이터 타입 확인하기

1. 다음 값을 각각 변수에 저장하세요:
    - 정수 42
    - 부동 소수점 수 3.14
    - 복소수 1 + 2j
2. 각 변수의 값을 출력하고, 데이터 타입도 함께 출력하세요.

출력 예시:

- 42 는 형식입니다.
- 3.14 는 형식입니다.
- (1+2j) 는 형식입니다.

문제 1: 숫자형 데이터 타입 확인하기 해답

```
# 숫자형 데이터 타입 확인하기
int_var = 42
float_var = 3.14
complex_var = 1 + 2j

print(int_var, "는", type(int_var), "형식입니다.")
```

```
## 42 는 <class 'int'> 형식입니다.
```

```
print(float_var, "는", type(float_var), "형식입니다.")
```

```
## 3.14 는 <class 'float'> 형식입니다.
```

```
print(complex_var, "는", type(complex_var), "형식입니다.")
```

문제 2: 문자열 연산

1. 변수 greeting에 "안녕"을 저장하세요.
2. 변수 name에 "파이썬"을 저장하세요.
3. greeting과 name을 결합하여 "안녕 파이썬!"이라는 문장을 출력하세요.
4. "파이썬"이라는 단어를 3번 반복하여 "파이썬파이썬파이썬"을 출력하세요.

문제 2: 문자열 연산

```
# 문자열 연산
greeting = "안녕"
name = "파이썬"

# 문자열 결합
message = greeting + " " + name + "!"
print(message)
```

```
## 안녕 파이썬!
```

```
# 문자열 반복
repeated = name * 3
print(repeated)
```

```
## 파이썬파이썬파이썬
```

문제 3: 리스트 생성 및 요소 접근

1. 다음 요소를 포함하는 리스트 fruits를 생성하세요: "apple", "banana", "cherry".
2. fruits 리스트의 첫 번째 요소와 마지막 요소를 출력하세요.
3. 리스트의 두 번째 요소를 "grape"로 변경한 후 전체 리스트를 출력하세요.

문제 3: 리스트 생성 및 요소 접근 해답

```
# 리스트 생성 및 요소 접근
fruits = ["apple", "banana", "cherry"]

# 첫 번째 요소와 마지막 요소 출력
print("첫 번째 요소:", fruits[0])
```

```
## 첫 번째 요소: apple
```

```
print("마지막 요소:", fruits[-1])
```

```
## 마지막 요소: cherry
```

```
# 두 번째 요소 변경
fruits[1] = "grape"
print("변경된 리스트:", fruits)
```

문제 4: 딕셔너리 값 가져오기

1. 다음 데이터를 포함하는 딕셔너리 person을 생성하세요:
    - 키 name의 값은 "Alice"
    - 키 age의 값은 25
    - 키 city의 값은 "Seoul"
2. person 딕셔너리에서 이름, 나이, 도시를 각각 출력하세요.
3. get() 메서드를 사용하여 "name"의 값을 출력하세요.

문제 4: 딕셔너리 값 가져오기 해답

```
# 딕셔너리 값 가져오기
person = {
    "name": "Alice",
    "age": 25,
    "city": "Seoul"
}

# 각 키의 값 출력
print("이름:", person["name"])
```

```
## 이름: Alice
```

```
print("나이:", person["age"])
```

```
## 나이: 25
```

문제 5: 집합 연산

1. 다음 두 개의 집합을 생성하세요:

- set1 = {"apple", "banana", "cherry"}
- set2 = {"banana", "cherry", "date", "fig"}

1. 두 집합의 교집합, 합집합, **차집합(set1에서 set2를 뺀 결과)**을 각각 출력하세요.

출력 예시:

- 교집합: {'banana', 'cherry'}
- 합집합: {'apple', 'banana', 'cherry', 'date', 'fig'}
- 차집합: {'apple'}

문제 5: 집합 연산 해답

```
# 집합 연산
set1 = {"apple", "banana", "cherry"}
set2 = {"banana", "cherry", "date", "fig"}

# 교집합
intersection = set1.intersection(set2)
print("교집합:", intersection)
```

```
## 교집합: {'cherry', 'banana'}
```

```
# 합집합
union = set1.union(set2)
print("합집합:", union)
```

```
## 합집합: {'date', 'fig', 'cherry', 'apple', 'banana'}
```