Python 기초

1. List

- 순서가 있고, 중복을 허용하며, 수정 가능한(mutable) 원소들의 모음
- 용도: 데이터의 순차적인 저장, 정렬, 필터링 등 순서가 중요한 데이터를 다룰 때 사용
- 주의사항: 리스트 크기가 커지면, 원소 추가 삭제 시 성능 저하 발생
- 코드 예시:

```
# 리스트 생성
my_list = [1, 2, 3, "apple", 3.14]

# 원소 추가
my_list.append("banana") # [1, 2, 3, 'apple', 3.14, 'banana']

# 원소 접근 (인덱싱)
print(my_list[0]) # 1

# 슬라이싱
print(my_list[1:3]) # [2, 3]

# 원소 삭제
my_list.remove(3) # [1, 2, 'apple', 3.14, 'banana']

# 리스트 컴프리헨션 (List Comprehension)
squares = [x**2 for x in range(5)] # [0, 1, 4, 9, 16]
```

2. Dictionary

- Key-Value 쌍으로 이루어진 순서가 없는(Python 3.7+ 부터는 입력 순서 유지) 데이터 구조
- 용도: Kev를 이용해 데이터에 빠르게 접근해야 할 때 사용, JSON 형식 데이터 다룰 때 유용
- **주의사항**: Key는 해시 가능(hashable), 고유 → 리스트나 다른 딕셔너리처럼 변경 가능한 객체는 Key가 될 수 없음
- 코드 예시:

```
# 딕셔너리 생성
my_dict = {"name": "Alice", "age": 25, "city": "New York"}

# 원소 추가/수정
my_dict["email"] = "alice@example.com"
my_dict["age"] = 26

# 원소 접근
print(my_dict["name"]) # "Alice"
print(my_dict.get("country", "USA")) # 키가 없으면 기본값 "USA" 반환
# 원소 삭제
```

```
del my_dict["city"]

# Key, Value, Item 순회

for key, value in my_dict.items():
    print(f"{key}: {value}")
```

• get() 메서드는 Key가 없을 때 오류 대신 기본값을 반환하여 안정적인 코드 작성 가능

3. Set

- 순서가 없고, 중복을 허용하지 않는(unique) 원소들의 모음
- 용도: 중복 없는 데이터 컬렉션을 만들거나, 원소 존재 여부 확인, 집합 연산을 수행할 때 사용
- 주의사항: 순서가 보장되지 않으므로, 인덱싱을 통해 원소에 접근할 수 없음
- 코드 예시:

```
# 세트 생성

my_set = {1, 2, 3, 2, 1} # {1, 2, 3}
list_to_set = set([1, "apple", 1]) # {1, 'apple'}

# 원소 추가

my_set.add(4) # {1, 2, 3, 4}

# 원소 삭제

my_set.remove(2) # {1, 3, 4}. 원소가 없으면 KeyError 발생

my_set.discard(10) # 원소가 없어도 에러 발생 안 함

# 집합 연산

set_a = {1, 2, 3}

set_b = {3, 4, 5}

print(set_a | set_b) # 합집합: {1, 2, 3, 4, 5}

print(set_a & set_b) # 교집합: {3}

print(set_a - set_b) # 차집합: {1, 2}
```

• remove()와 discard()의 차이점을 이해하는 것이 중요

4. Tuple

- 순서가 있고, 중복을 허용하지만, 수정 불가능한(immutable) 원소들의 모음
- 용도: 함수에서 여러 값 반환, 변경되지 않아야 하는 데이터의 묶음(딕셔너리의 키)을 표현할 때 사용
- 주의사항: 생성 후에는 원소를 추가, 수정, 삭제할 수 없음
- 코드 예시:

```
# 튜플 생성
my_tuple = (1, "hello", 3.14)
single_tuple = (1,) # 원소가 하나일 때는 쉼표(,)가 필수
# 원소 접근 (인덱싱)
print(my_tuple[1]) # "hello"
```

```
# 패킹과 언패킹
packed_tuple = 1, 2, "world" # 패킹
a, b, c = packed_tuple # 언패킹
print(b) # 2

# 함수에서 여러 값 반환
def get_min_max(numbers):
    return min(numbers), max(numbers)

min_val, max_val = get_min_max([1, 5, 2, 8])
print(f"Min: {min_val}, Max: {max_val}") # Min: 1, Max: 8
```

5. 함수 (Function)

- 특정 작업을 수행하는 코드 블록으로, 재사용이 가능
- 용도: 코드의 재사용성을 높이고, 로직을 모듈화, 가독성과 유지보수성 향상 시 사용
- 주의사항: 함수는 하나의 기능만 수행하도록 작성(단일 책임 원칙)/함수 인자의 기본값은 변경 불가능한 (immutable) 객체를 사용하는 것이 안전
- 코드 예시:

```
# 기본 함수 정의
def greet(name):
    return f"Hello, {name}!"
# 기본값이 있는 함수
def calculate_area(width, height=10):
    return width * height
# 가변 인자를 받는 함수
def sum all(*args):
   total = 0
   for num in args:
       total += num
    return total
# 키워드 가변 인자를 받는 함수
def print info(**kwargs):
    for key, value in kwargs.items():
       print(f"{key}: {value}")
print(greet("World")) # Hello, World!
print(calculate_area(5)) # 50
print(sum_all(1, 2, 3, 4)) # 10
print_info(name="Bob", age=30) # name: Bob, age: 30
```

- *args는 임의의 개수의 위치 인자를 튜플로 받음
- **kwargs는 임의의 개수의 키워드 인자를 딕셔너리로 받음

데코레이터

- 함수는 데코레이터로 감싸서, 기존 함수를 유지하며 새로운 기능을 추가 가능
- 핵심: *args, **kwargs를 사용해 범용성을 확보하고, 원래 함수의 return 값을 처리해야 함
- 권장사항: functools.wraps를 사용해 원래 함수의 메타데이터를 보존 (없을 경우 메타데이터가 데코레이터 정보로 덮어씌워짐)
- 코드 예시:

```
import time
from functools import wraps
def timer_decorator_fixed(function):
   @wraps(function) # 이 부분을 추가!
   def wrapper(*args, **kwargs):
       start_time = time.time()
       result = function(*args, **kwargs)
       end time = time.time()
       print(f"'{function.__name__}' 함수 실행 시간: {end_time -
start_time:.4f}초")
       return result
   return wrapper
@timer_decorator_fixed
def add fixed(a, b):
   """두 숫자를 더하는 함수입니다."""
   return a + b
print(add_fixed.__name__) # 'add_fixed' 출력
print(add_fixed.__doc__) # '두 숫자를 더하는 함수입니다.' 출력
```

6. 클래스 (Class)

- 객체를 생성하기 위한 틀(template)로, 속성(attribute)과 메서드(method)를 가짐
- 용도: 객체 지향 프로그래밍을 통해 데이터와 관련 동작을 캡슐화
- 주의사항: 상속과 다형성을 적절히 활용하면 유연하고 확장성 있는 코드를 작성 가능
- 코드 예시:

```
class Dog:
# 클래스 변수
species = "Canis lupus familiaris"

# 초기화 메서드 (생성자)
def __init__(self, name, age):
# 인스턴스 변수
self.name = name
self.age = age

# 인스턴스 메서드
def bark(self):
```

```
return "Woof!"

def get_info(self):
    return f"{self.name} is {self.age} years old."

# 객체(인스턴스) 생성

my_dog = Dog("Buddy", 3)

# 메서드 호출

print(my_dog.get_info()) # Buddy is 3 years old.

print(my_dog.bark()) # Woof!

# 속성 접근

print(my_dog.name) # Buddy

print(Dog.species) # Canis lupus familiaris
```

- __init__ 메서드는 객체가 생성될 때 호출
- self는 생성된 인스턴스 자신을 가리킴

장단점 및 대안

자료구조/ 개념	장점	단점	대안
List	순서 유지, 유연한 데이터 조작	대용량 데이터에서 추가/삭제 시 성능 저하	collections.deque (양방향 큐, 빠른 추가/삭 제), numpy.array (수치 계산 특화)
Dictionary	빠른 Key 기반 조 회 (O(1) 시간 복잡 도)	Key는 해시 가능해 야 함, List보다 메 모리 사용량이 많 음	collections.defaultdict (기본값 자동 생성), collections.OrderedDict (순서 보장, Python 3.7+ 부터는 기본 dict도 순서 유지)
Set	빠른 원소 검색, 중 복 자동 제거, 효율 적인 집합 연산	순서가 없어 인덱 싱 불가	frozenset (수정 불가능한 set)
Tuple	불변성으로 데이터 안정성 보장, List보 다 메모리 효율적	생성 후 수정 불가	collections.namedtuple (이름으로 원소 접근 가능), List (수정이 필요할 경우)
함수	코드 재사용성, 가 독성, 유지보수성 향상	잘못 설계하면 오 히려 복잡도 증가	람다(lambda) 함수 (간단한 익명 함수 필요 시)
클래스	데이터와 로직 캡 슐화, 코드의 구조 화 및 확장성	간단한 작업에는 과도한 설계가 될 수 있음	간단한 경우, 함수와 딕셔너리 조합으로 대체 가 능