

데이터프로그래밍 기초 4일차

2026-1 DS Bootcamp

부산대학교
데이터사이언스전문대학원
석사과정 박민서

CONTENTS

1 Object Oriented Programming

2 Class

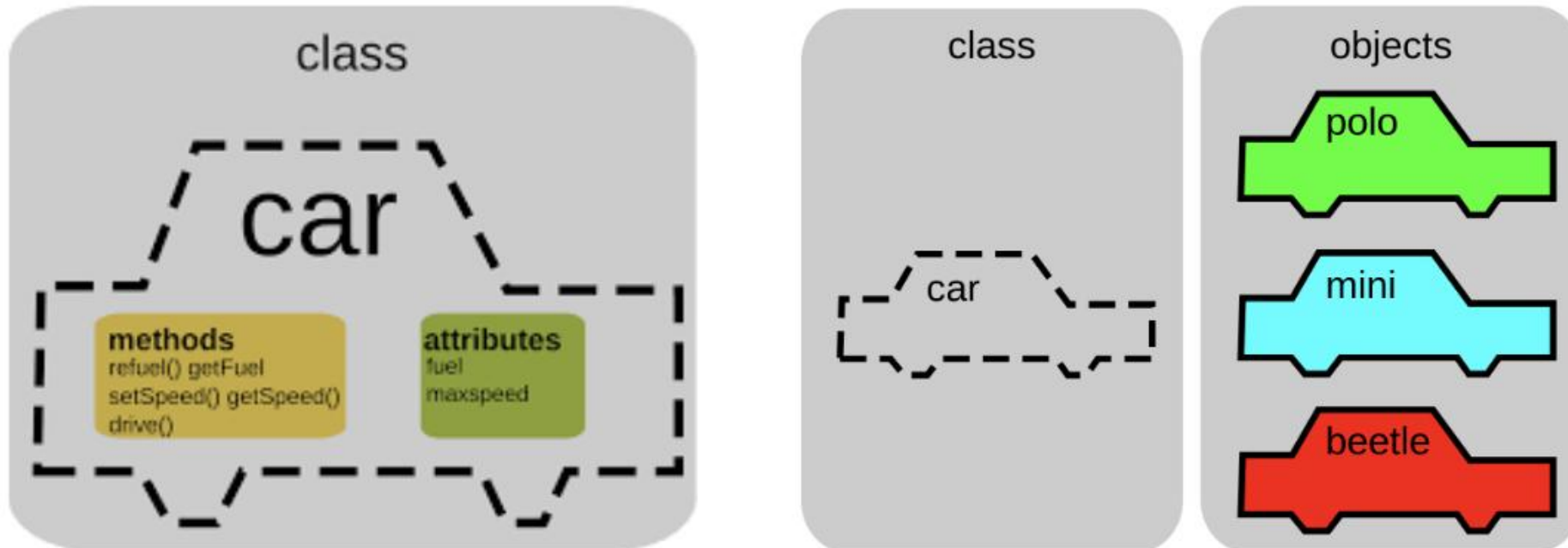
3 Class Inheritance

4 Practice

5 Homework

■ 객체 지향 프로그래밍(OOP, Object Oriented Programming)

- 객체: 프로그램 내에서 속성(attributes)을 가질 수 있는 모든 데이터
- 객체 지향 프로그래밍: 객체를 우선으로 생각하는 프로그래밍



■ 객체 지향 프로그래밍(OOP, Object Oriented Programming)

- 객체: 프로그램 내에서 속성(attributes)을 가질 수 있는 모든 데이터
- 객체 지향 프로그래밍: 객체를 우선으로 생각하는 프로그래밍

```
# students 속 각 학생들의 이름 및 과목별 성적 저장
# 각 학생(student)은 "name", "korean", ... 등의 속성을 가짐 => '객체(object)'
students = [
    {"name": "윤인성", "korean": 87, "math": 98, "english": 88, "science": 95},
    {"name": "연하진", "korean": 92, "math": 98, "english": 96, "science": 98},
    {"name": "구지연", "korean": 76, "math": 96, "english": 94, "science": 90},
    {"name": "나선주", "korean": 98, "math": 92, "english": 96, "science": 92},
    {"name": "윤아린", "korean": 95, "math": 98, "english": 98, "science": 98},
    {"name": "윤명월", "korean": 64, "math": 88, "english": 92, "science": 92},
]

# 학생 수만큼 반복
print("이름", "총점", "평균", sep="\t")

for student in students:
    score_sum = student["korean"] + student["math"] + \
        student["english"] + student["science"]
    score_average = score_sum / 4
    print(student["name"], score_sum, score_average, sep="\t")
```

이름	총점	평균
윤인성	368	92.0
연하진	384	96.0
구지연	356	89.0
나선주	378	94.5
윤아린	389	97.25
윤명월	336	84.0

■ 객체 지향 프로그래밍(OOP, Object Oriented Programming)

- 객체: 프로그램 내에서 속성(attributes)을 가질 수 있는 모든 데이터
- 객체 지향 프로그래밍: 객체를 우선으로 생각하는 프로그래밍

```
# student란 객체와 관련된 코드
def create_student(name, korean, math, english, science):
    return {
        "name": name,
        "korean": korean,
        "math": math,
        "english": english,
        "science": science
    }

def student_get_sum(student):
    return student["korean"] + student["math"] + \
        student["english"] + student["science"]

def student_get_average(student):
    return student_get_sum(student) / 4

def student_to_string(student):
    return f"{student["name"]}\t{student_get_sum(student)}\t{student_get_average(student)}"
```

```
# student란 객체를 활용하는 코드
students = [
    create_student("윤인성", 87, 98, 88, 95),
    create_student("연하진", 92, 98, 96, 98),
    create_student("구지연", 76, 96, 94, 90),
    create_student("나선주", 98, 92, 96, 92),
    create_student("윤아린", 95, 98, 98, 98),
    create_student("윤명월", 64, 88, 92, 92)
]

print("이름", "총점", "평균", sep="\t")

for student in students:
    print(student_to_string(student))
```

■ 클래스 (Class)

- 객체의 구조 및 동작을 정의하는 구문 ≡ ‘사용자 정의 자료형’
- **class 클래스 이름:**
 클래스와 관련된 코드
- 인스턴스 (Instance): Class 기반으로 만들어진 객체
- **인스턴스 이름(변수) = 클래스 이름() [= 생성자 함수]**

```
class Student:
    pass
    # 여기에 Student 클래스 구현 코드

# 학생 instance 선언
student = Student()
```

■ 생성자 (Constructor)

- 객체가 생성될 때, 자동으로 처리되는 명령 모음
- 클래스 내부에서 “__init__(self, 추가 매개변수)” 형태로 정의됨
- “self”: 생성된 객체를 참조할 때 (= 객체의 속성이나 기능에 접근할 때) 사용

```
class Student:
    def __init__(self, name, korean, math, english, science):
        # 객체 생성 시,
        self.name = name # 입력받은 name을 객체의 name 속성에 저장
        self.korean = korean
        self.math = math
        self.english = english
        self.science = science
```

```
# Student의 instance 생성
Kim = Student("Kim", 87, 98, 88, 95)
# Student instance의 속성에 접근
print(Kim.name)
print(Kim.korean)
```

✓ 0.0s

Kim
87

■ 소멸자 (Destructor)

- 객체가 '삭제'될 때, 자동으로 처리되는 명령 모음
- 클래스 내부에서 “`__del__(self, 추가 매개변수)`” 형태로 정의됨

```
class Account:
    num_accounts = 0
    # 생성자
    def __init__(self, name):
        self.name = name
        print(f"{name}의 계좌가 생성되었습니다.")
        Account.num_accounts += 1
    # 소멸자
    def __del__(self):
        Account.num_accounts -= 1
        print("계좌 삭제가 완료되었습니다.")

member1 = Account("Park") # 생성자 __init__ 실행
del member1 # 소멸자 __del__ 실행
```

✓ 0.0s

Park의 계좌가 생성되었습니다.
계좌 삭제가 완료되었습니다.

■ 메서드 (Method)

- 클래스가 가지고 있는 함수

- class 클래스 이름:

def 메소드 이름(self, 추가 매개변수):

실행할 코드

```
class Student:
    def __init__(self, name, korean, math, english, science):
        # 객체 생성 시,
        self.name = name # 입력받은 name을 객체의 name 속성에 저장
        self.korean = korean
        self.math = math
        self.english = english
        self.science = science

    # Student 클래스 속 과목 점수 총합 반환하는 method
    def get_sum(self):
        return self.korean + self.math + self.english + self.science

    # Student 클래스 속 과목 점수 평균 반환하는 method
    def get_average(self):
        return self.get_sum() / 4

    # Student 클래스 속 이름, 총합, 평균이 있는 문자열 반환 method
    def to_string(self):
        return f"{self.name}\t{self.get_sum()}\t{self.get_average()}"
```

```
print("이름", "총점", "평균", sep="\t")
```

```
students = [
    Student("윤인성", 87, 98, 88, 95),
    Student("연하진", 92, 98, 96, 98),
    Student("구지연", 76, 96, 94, 90),
    Student("나선주", 98, 92, 96, 92),
    Student("윤아린", 95, 98, 98, 98),
    Student("윤명월", 64, 88, 92, 92)
]
```

```
for student in students:
    print(student.to_string())
```

✓ 0.0s

이름	총점	평균
윤인성	368	92.0
연하진	384	96.0
구지연	356	89.0
나선주	378	94.5
윤아린	389	97.25
윤명월	336	84.0

■ 클래스 속 변수

- 클래스 변수: 클래스 내부에서 선언된 변수
- 인스턴스 변수: “self”가 붙은 변수

```
class Dog:
    species = "강아지" # 클래스 변수

    def __init__(self, name):
        self.name = name # 인스턴스 변수

d1 = Dog("초코")
d2 = Dog("바둑이")

print(d1.species)
print(d2.species)
# 클래스 변수 변경 -> 모든 instance의 클래스 변수 값이 변경
Dog.species = "Dog"
print(d1.species) # 변경된 "Dog"로 출력
print(d2.species) # 변경된 "Dog"로 출력

✓ 0.0s
```

강아지
강아지
Dog
Dog

■ 클래스 속 변수

- 클래스 변수: 클래스 내부에서 선언된 변수
- 인스턴스 변수: “self”가 붙은 변수

```
class Dog:
    species = "강아지" # 클래스 변수

    def __init__(self, name):
        self.name = name # 인스턴스 변수

d1 = Dog("초코")
d2 = Dog("바둑이")

print(d1.species) # "강아지"
print(d2.species)

# instance의 클래스 변수 변경
d1.species = "Dog"

print(d1.species) # "Dog"
print(d2.species) # "강아지"

✓ 0.0s
```

강아지
강아지
Dog
강아지

▪ 특수한 이름의 메소드

- “__str__”: 인스턴스를 print()를 이용해 출력할 때 출력할 내용 반환

```
class Student:
    def __init__(self, name, korean, math, english, science):
        # 객체 생성 시,
        self.name = name # 입력받은 name을 객체의 name 속성에 저장
        self.korean = korean
        self.math = math
        self.english = english
        self.science = science

    def get_sum(self):
        return self.korean + self.math + self.english + self.science

    def get_average(self):
        return self.get_sum() / 4

    def __str__(self):
        return f"{self.name}\t{self.get_sum()}\t{self.get_average()}"

    # def to_string(self):
    #     return f"{self.name}\t{self.get_sum()}\t{self.get_average()}"
```

```
print("이름", "총점", "평균", sep="\t")

students = [
    Student("윤인성", 87, 98, 88, 95),
    Student("연하진", 92, 98, 96, 98),
    Student("구지연", 76, 96, 94, 90),
    Student("나선주", 98, 92, 96, 92),
    Student("윤아린", 95, 98, 98, 98),
    Student("윤명월", 64, 88, 92, 92)
]

for student in students:
    # print(student.to_string())
    print(student)
```

이름	총점	평균
윤인성	368	92.0
연하진	384	96.0
구지연	356	89.0
나선주	378	94.5
윤아린	389	97.25
윤명월	336	84.0

■ 특수한 이름의 메소드

- “`__eq__`”: 같은 인스턴스끼리 ‘`==`’ 연산자를 이용해 비교하는 경우
- “`__ne__`”: 같은 인스턴스끼리 ‘`!=`’ 연산자를 이용해 비교하는 경우

```
class Student:
    def __init__(self, name, korean, math, english, science):
        # 객체 생성 시,
        self.name = name # 입력받은 name을 객체의 name 속성에 저장
        self.korean = korean
        self.math = math
        self.english = english
        self.science = science

    def get_sum(self):
        return self.korean + self.math + self.english + self.science

    # '=='를 이용해 비교할 때
    # 'value'는 '==' 연산자 뒤에 올 변수를 칭함
    def __eq__(self, value):
        print("총점이 같습니다.")
        return self.get_sum() == value.get_sum()

    # '!='를 이용해 비교할 때
    def __ne__(self, value):
        print("총점이 다릅니다.")
        return self.get_sum() != value.get_sum()
```

```
s1 = Student("윤인성", 87, 98, 88, 95)
s2 = Student("연하진", 92, 98, 96, 98)

print(s1 == s2) # __eq__
print(s1 != s2) # __ne__
```

```
총점이 같습니다.
False
총점이 다릅니다.
True
```

■ 특수한 이름의 메소드

- “__gt__”: 같은 인스턴스끼리 ‘>’ 연산자를 이용해 비교하는 경우
- “__ge__”: 같은 인스턴스끼리 ‘>=’ 연산자를 이용해 비교하는 경우

```
class Student:
    def __init__(self, name, korean, math, english, science):
        # 객체 생성 시,
        self.name = name # 입력받은 name을 객체의 name 속성에 저장
        self.korean = korean
        self.math = math
        self.english = english
        self.science = science

    def get_sum(self):
        return self.korean + self.math + self.english + self.science

    # '>'를 이용해 비교할 때
    def __gt__(self, value):
        print(f"{self.name}의 총점이 {value.name}의 총점보다 높습니다.")
        return self.get_sum() > value.get_sum()

    # '>='를 이용해 비교할 때
    def __ge__(self, value):
        print(f"{self.name}의 총점이 {value.name}의 총점 이상입니다.")
        return self.get_sum() >= value.get_sum()
```

```
s1 = Student("윤인성", 87, 98, 88, 95)
s2 = Student("연하진", 92, 98, 96, 98)

print(s1 > s2) # __gt__
print(s1 >= s2) # __ge__
```

```
윤인성의 총점이 연하진의 총점보다 높습니다.
False
윤인성의 총점이 연하진의 총점 이상입니다.
False
```

■ 특수한 이름의 메소드

- “__lt__”: 같은 인스턴스끼리 ‘<’ 연산자를 이용해 비교하는 경우
- “__le__”: 같은 인스턴스끼리 ‘<=’ 연산자를 이용해 비교하는 경우

```
class Student:
    def __init__(self, name, korean, math, english, science):
        # 객체 생성 시,
        self.name = name # 입력받은 name을 객체의 name 속성에 저장
        self.korean = korean
        self.math = math
        self.english = english
        self.science = science

    def get_sum(self):
        return self.korean + self.math + self.english + self.science

    # '<'를 이용해 비교할 때
    def __lt__(self, value):
        print(f"{self.name}의 총점이 {value.name}의 총점보다 낮습니다.")
        return self.get_sum() < value.get_sum()

    # '<='를 이용해 비교할 때
    def __le__(self, value):
        print(f"{self.name}의 총점이 {value.name}의 총점 이하입니다.")
        return self.get_sum() <= value.get_sum()
```

```
s1 = Student("윤인성", 87, 98, 88, 95)
s2 = Student("연하진", 92, 98, 96, 98)

print(s1 < s2) # __lt__
print(s1 <= s2) # __le__
```

```
윤인성의 총점이 연하진의 총점보다 낮습니다.
True
윤인성의 총점이 연하진의 총점 이하입니다.
True
```

■ 실습 1

- 알바들의 정보가 담긴 'parttimer.txt' 파일을 읽고, 클래스 'PartTimer'를 다음을 참고하여 작성하세요.
 - 생성시, 이름 (str), 시급 (정수), 근무시간(정수, 초기값 0)가 초기화
 - 메서드 work(): 근무시간 누적, 근무시간은 양수로 주어진다고 가정
 - 메서드 get_salary(): 누적된 급여를 반환, 급여 = 누적시간 × 시급
 - 누적된 급여를 기준으로 정렬한 뒤, 상위 3명의 알바에 대한 정보를 출력
 - 객체 출력 시, 다음과 같은 형식으로 출력
“이름: *** / 근무시간: **시간 / 급여: **원”

급여 상위 3명:

이름: Seoyeon / 근무시간: 15시간 / 급여: 135000원

이름: Junho / 근무시간: 10시간 / 급여: 125000원

이름: Hyunwoo / 근무시간: 11시간 / 급여: 121000원

```
민서,10000,5,3,2
지훈,12000,4,4,1
서연,9000,8,4,3
현우,11000,6,2,3
유진,9500,7,3,2
도윤,13000,3,4,2
수아,10500,5,5,1
민재,11500,4,3,3
지아,9800,6,4,2
준호,12500,2,5,3
```

parttimer.txt

Q&A

■ 클래스 상속(Inheritance)

- 이미 구현된 클래스의 속성과 메서드를 그대로 사용할 수 있는 기능
- class 상속받을 클래스 이름(상속할 클래스):

관련 코드

```
class Animal:
    def __init__(self, name):
        self.name = name

    def speak(self):
        print("동물이 소리를 냅니다.")

# Animal 클래스를 상속받은 Dog 클래스 정의
class Dog(Animal):
    # 부모 클래스 Animal의 메서드 speak()를 덮어쓰기 (override)
    def speak(self):
        print(f"{self.name}: 멍멍!")

# Animal 클래스를 상속받은 Cat 클래스 정의
class Cat(Animal):
    def speak(self):
        print(f"{self.name}: 야옹!")
```

```
a = Animal("알수없음")
d = Dog("초코")
c = Cat("나비")

a.speak()
d.speak()
c.speak()
```

```
동물이 소리를 냅니다.
초코: 멍멍!
나비: 야옹!
```

■ 클래스 상속(Inheritance)

- `super()`: 상속받은 클래스에 접근하기 위한 방법

```
class Animal:
    def __init__(self, name):
        self.name = name

class Dog(Animal):
    def __init__(self, name, breed):
        self.breed = breed

d = Dog("Buddy", "Poodle")
print(d.name)  # 에러 발생, Dog 클래스에서 name이 초기화되지 않음
```

```
AttributeError                                Traceback (most recent call last)
Cell In[21], line 10
      7         self.breed = breed
      9 d = Dog("Buddy", "Poodle")
--> 10 print(d.name)  # 에러 발생, Dog 클래스에서 name이 초기화되지 않음

AttributeError: 'Dog' object has no attribute 'name'
```

```
class Animal:
    def __init__(self, name):
        self.name = name

class Dog(Animal):
    def __init__(self, name, breed):
        # 상속받은 Animal 클래스에서 초기화한 대로
        # name 변수를 입력받아 초기화
        super().__init__(name)
        self.breed = breed

d = Dog("Buddy", "Poodle")
print(d.name)  # "Buddy"
```

✓ 0.0s

Buddy

■ 실습 2

- 은행 계좌 관리하는 클래스들을 다음을 참고하여 작성하세요.

- Account 클래스

- 생성 시, owner(문자열), balance(정수)가 초기화 됨
- deposit(amount): amount만큼 입금하는 메서드
- withdraw(amount): amount만큼 출금하는 메서드
- 객체 출력 시, 다음과 같은 형식으로 출력
 - 소유자: *** / 잔액: ***원.

- SavingsAccount 클래스 (Account 클래스 상속 받음)

- interest_rate(실수)도 추가로 초기화 (0~1 사이의 실수)
- apply_interest(): 현재 잔액에 이자를 적용하는 메서드

- CheckingAccount 클래스 (Account 클래스 상속 받음)

- limit(정수)도 추가로 초기화 (음의 정수)
- withdraw(amount): 잔액이 '- limit'를 벗어나는 출금이면 한도 초과 메시지 띄우는 메서드

```
a = SavingsAccount("Minsuh", 100000, 0.05)
b = CheckingAccount("Jihoon", 50000, 30000)
```

```
print(a)
print(b)
```

```
a.deposit(20000)
a.apply_interest()
print(a)
```

```
b.withdraw(70000)
print(b)
```

```
b.withdraw(20000)
print(b)
```

✓ 0.0s

```
소유자: Minsuh / 잔액: 100000원
소유자: Jihoon / 잔액: 50000원
소유자: Minsuh / 잔액: 126000원
소유자: Jihoon / 잔액: -20000원
소유자: Jihoon / 잔액: -20000원
```

▪ 6일차 과제

- GitHub 사이트에서 “6일차_과제.ipynb” 다운로드
- 코드 작성 후, “**본인이름**_6일차_과제.ipynb”로 저장
- 저장한 과제 파일 전송 (이메일 주소: minsuh99@pusan.ac.kr)
기한: ~ 2/10 PM 13:59:59

Q&A
