codingOn X posco

K-Digital Training 스마트팩토리 개발자 입문

codingOn

Python 클래스

학습목표

- 상속에 대해서 이해한다.
- 다형성에 대해서 이해한다.
- 추상화에 대해서 이해한다.
- 클래스메서드와 정적메서드에 대해서 이해한다.

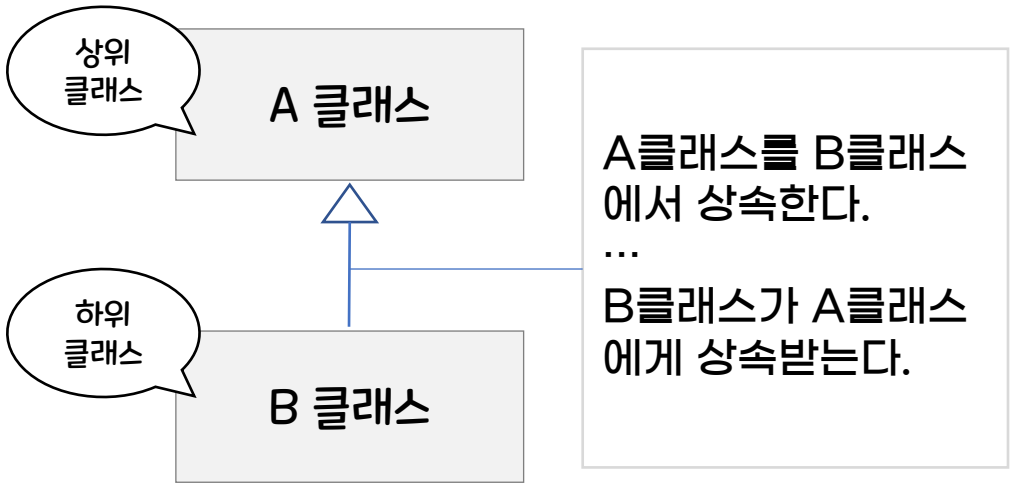
codingOn

상속

상속(Inheritance)

- **상속이란?**
  - 클래스를 정의할때 이미 구현된 클래스를 상속(inheritance) 받아서 속성이나 기능이 확장되는 클래스를 구현함.

```
class 클래스 이름(상속할 클래스 이름)
```



상속

- 기존의 클래스가 가지고 있는 필드와 메서드를 그대로 물려받는
  새로운 클래스를 만드는 것
- 새로운 클래스에서 필드, 메서드 추가 가능
- 공통점을 가진 내용을 상위 클래스에 둬서 일관되고 효율적인 코딩 가능

상속

- 부모(슈퍼) 클래스 , 자식(서브) 클래스
  - 예) 국가 클래스 (부모), 한국,일본,중국 (자식)
    예) 동물 클래스 (부모), 개,고양이,호랑이(자식)
- 자식 클래스를 선언할 때 소괄호로 부모 클래스 포함시키기
- super() : 자식 클래스에서 부모 클래스의 값 사용할 때 쓰는 키워드

```
class Parent:
  # Parent 클래스 코드

class Child(Parent):
  # Child 클래스 코드
```

상속

- 부모클래스에 생성자가 없는 경우
- 생성자가 없는 부모 클래스를 상속받을 때 자식 클래스도 별도의 생성자를 구현하지 않아도 부모 클래스의 속성과 메서드를 바로 사용

```python
# 부모 클래스: 생성자 없음
class Animal:
    def speak(self):
        print("동물이 소리를 냅니다.")

    def move(self):
        print("동물이 움직입니다.")

# 자식 클래스
# Animal 클래스를 상속
class Dog(Animal):
    def bark(self):
        print("멍멍!")

# 객체 생성
dog = Dog()
dog.speak()  # 동물이 소리를 냅니다.
dog.move()   # 동물이 움직입니다.
dog.bark()   # 멍멍!
```

상속

- 부모클래스에 생성자가 있는 경우
- 자식 클래스에서 부모 클래스의 생성자를 super().__init__()으로 명시적으로 호출, self는 필요없음
- 자식클래스의 생성자 매개변수를 꼭 부모클래스와 동일하게 안해도 되지만 사용시에는 self.변수명으로 사용

```python
# 부모 클래스: 생성자 있음
class Animal:
    def __init__(self, name):
        self.name = name

    def speak(self):
        print(f"{self.name}가 소리를 냅니다.")

    def move(self):
        print(f"{self.name}가 움직입니다.")

# 자식 클래스
# Animal 클래스를 상속
class Dog(Animal):
    def __init__(self, name, sound):
        super().__init__(name)  # super()로 부모클래스 생성자 호출
        self.sound = sound  # 자식 클래스 속성 추가

    def bark(self):
        print(f"{self.name}가 {self.sound} 짖습니다.")

# 객체 생성
dog = Dog("백구", "멍멍")
dog.speak()  # 백구가 소리를 냅니다.
dog.move()   # 백구가 움직입니다.
dog.bark()   # 백구가 멍멍 짖습니다.
```

다중상속

codingOn

- 파이썬은 하나의 자식 클래스가 여러 부모 클래스를 상속 받을 수 있음
- 클래스명.__init__() 으로 명시적 호출, self가 필요함

```python
class Engine:
    def __init__(self, horsepower):
        self.horsepower = horsepower

class Wheels:
    def __init__(self, wheel_count):
        self.wheel_count = wheel_count

class Car(Engine, Wheels):
    def __init__(self, horsepower, wheel_count):
        Engine.__init__(self, horsepower)  # 첫 번째 부모 생성자 호출
        Wheels.__init__(self, wheel_count)  # 두 번째 부모 생성자 호출

    def info(self):
        print(f"이 자동차는 {self.horsepower} 마력 엔진과 {self.wheel_count}개의 바퀴를 가지고 있습니다.")

car = Car(150, 4)
car.info()  # 이 자동차는 150 마력 엔진과 4개의 바퀴를 가지고 있습니다.
print(Car.mro()) # mro()를 사용하여 클래스의 상속 순서 리스트를 반환
```

다형성

다형성

- 다형성은 동일한 이름의 메서드가 객체의 타입에 따라 다르게 동작하는 것을 말함
- 주로 상속 관계에서 메서드 동작을 변경
- 다형성을 실현하는 기법중 하나로 오버라이딩이 존재

오버라이딩

- 메서드 재정의(Method Overriding)
- 상속된 메서드의 내용이 자식 클래스에 맞지 않을 경우, 자식 클래스에서 동일한 메서드를 재정의 하는 것
- 즉, 자식 클래스는 부모 클래스의 기본 동작을 변경하거나 확장할 수 있음
- super().메서드명을 사용하여 자식 클래스에서 부모 클래스의 메서드를 호출할 수 있음
- 오버라이딩한 메서드는 부모 메서드의 매개변수 개수와 동일하게 사용하는 것을 권장

오버라이딩

```python
class Parent:
    def greet(self):
        print("안녕하세요, 저는 부모 클래스입니다.")

class Child(Parent):
    def greet(self):
        super().greet()  # 부모 클래스의 greet 호출
        print("안녕하세요, 저는 자식 클래스입니다.")

parent = Parent()
child = Child()
parent.greet()  # 안녕하세요, 저는 부모 클래스입니다.
print()
child.greet()
# 안녕하세요, 저는 부모 클래스입니다.
# 안녕하세요, 저는 자식 클래스입니다.
```

실습. 상속과 오버라이딩

codingOn

- 주어진 부모클래스를 바탕으로 조건을 만족하는 자식클래스 Electronic클래스와 Food클래스를 설계하여 보세요
- 부모클래스

```python
class Product:
    def __init__(self, name, price, quantity):
        self.name = name
        self.price = price
        self.quantity = quantity

    # 재고 업데이트 메서드
    def update_quantity(self, amount):
        self.quantity += amount
        print(f"{self.name} 재고가 {amount}만큼 {'증가' if amount > 0 else '감소'}했습니다. 현재 재고: {self.quantity}")

    # 상품 정보 출력 메서드
    def display_info(self):
        print(f"상품명: {self.name}")
        print(f"가격: {self.price}원")
        print(f"재고: {self.quantity}개")
```

실습. 상속과 오버라이딩

Electronic 클래스 설계조건
추가 변수: warranty_period (보증 기간, 기본값 12개월)
새로운 메서드: extend_warranty(months)를 작성하여 보증 기간을 연장
오버라이딩: display_info 메서드를 오버라이딩하여 보증 기간 정보를 포함한 상품 정보를 출력

Food 클래스 설계조건
추가 변수: expiration_date (유통기한 - 날짜형태는 "YYYY-MM-DD")
새로운 메서드: is_expired(current_date)를 작성하여 유통기한이 지났는지 여부를 확인하고 결과를 출력
오버라이딩: display_info 메서드를 오버라이딩하여 유통기한 정보를 포함한 상품 정보를 출력

```
상품명: 스마트 TV
가격: 1500000원
재고: 5개
보증 기간: 24개월
보증 기간이 12개월 연장되었습니다. 현재 보증 기간: 36개월
상품명: 스마트 TV
가격: 1500000원
재고: 5개
보증 기간: 36개월
```

```
사과는 유통기한이 지나지 않았습니다.
사과는 유통기한이 지났습니다.
상품명: 사과
가격: 3000원
재고: 50개
```

출력예시

추상화

추상화

- 추상화(Abstraction)
- 복잡한 시스템의 세부 사항을 감추고, 필요한 부분만 노출함으로써 코드를 더 간단하고 명확하게 작성
- 복잡한 구현 세부 사항을 몰라도 제공된 기능을 사용
- abc (Abstract Base Class) 모듈을 사용해 추상화를 지원

추상클래스

- 추상 클래스는 하나 이상의 추상 메서드를 포함한 클래스
- 구체적인 구현 없이 구조만 제공하며, 이를 기반으로 구체적인 동작을 구현하는 자식클래스를 설계하는 데 사용
- 직접 인스턴스화할 수 없으며, 반드시 상속받아 사용해야 함
- 추상 클래스는 생성자, 일반 메서드와 추상 메서드를 모두 포함 가능

즉, 공통된 동작은 추상 클래스에서 정의하고, 구체적인 동작은 자식클래스에서 구현

추상메서드

- 추상 메서드는 선언만 되어 있는 메서드
- 자식클래스에서 반드시 구현해야 하는 메서드로 강제성을 지님
- 추상 메서드는 추상 클래스 내에 정의되며, @abstractmethod 데코레이터를 사용
  - 데코레이터? 기존 함수나 메서드의 동작을 수정하거나 확장할 때 사용

추상클래스

```python
from abc import ABC, abstractmethod

# 추상 클래스 정의
class PaymentSystem(ABC):
    @abstractmethod
    def authenticate(self):
        pass

    @abstractmethod
    def process_payment(self, amount):
        pass

    def payment_summary(self, amount):
        print(f"{amount} 원 결제가 완료되었습니다.")

# 카드 결제 구현
class CreditCard(PaymentSystem):
    def authenticate(self):
        print("신용카드 인증 완료.")

    def process_payment(self, amount):
        print(f"신용카드로 {amount} 원을 결제합니다.")
```

```python
print("신용카드 결제:")
credit_card = CreditCard()
credit_card.authenticate()
credit_card.process_payment(50000)
credit_card.payment_summary(50000)
# 신용카드 결제:
# 신용카드 인증 완료.
# 신용카드로 50000 원을 결제합니다.
# 50000 원 결제가 완료되었습니다.
```

클래스 메서드와 정적 메서드

개념

- 메서드의 self 개념 중 클래스를 인스턴스화 하지 않고 호출해서 사용하는 경우에는 self가 필요 없음
- 위 경우에 일반적으로 클래스 메서드나 정적 메서드를 활용

클래스 메서드

- 클래스 메서드는 자기 자신의 클래스를 첫 번째 인자로 받는 메서드, cls 사용
- 클래스 속성에 접근하거나 클래스 상태를 변경할 수 있음
  - 클래스 변수에 접근하거나 수정할 때 사용
  - 클래스 또는 자식클래스 간의 동작을 통합적으로 처리 할때
- 인스턴스 생성 후 사용하는 것이 아닌 클래스에서 호출
- @classmethod 데코레이터 사용

클래스 메서드

```python
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    @classmethod
    def from_birth_year(cls, name, birth_year):
        age = 2024 - birth_year
        return cls(name, age)

# 클래스 메서드를 통해 객체 생성
p = Person.from_birth_year("홍길동", 1990)
print(p.name, p.age)  # "홍길동 34"
```

정적 메서드

- cls나 self를 사용하지 않음
- 클래스와 독립적인 작업(예: 유틸리티 함수)을 정의할 때 유용
- @staticmethod 데코레이터 사용
- 정적 메서드는 일반 함수로 작성해도 됨. 다만, 함수의 목적을 클래스와 연관 지어 코드의 효율성을 높일 수 있음

정적 메서드

```python
class MathUtils:
    @staticmethod
    def add(a, b):
        return a + b

    @staticmethod
    def multiply(a, b):
        return a * b

print(MathUtils.add(10, 20))       # 30
print(MathUtils.multiply(10, 20))  # 200
```

정적메서드 예시

```python
def add(a, b):
    return a + b

def multiply(a, b):
    return a * b

print(add(10, 20))  # 30
print(multiply(10, 20))  # 200
```

일반함수 예시

실습. 클래스 종합 프로그래밍

codingOn

- 아래 주어진 내용을 참고하여 프로그래밍을 진행하세요

```
#날짜별 전력사용량
electricity_usage = [
    {"date": "2024-11-01", "usage": 12.5},
    {"date": "2024-11-02", "usage": 15.3},
    {"date": "2024-11-03", "usage": 10.8},
    {"date": "2024-11-04", "usage": 14.2},
    {"date": "2024-11-05", "usage": 13.6}
]

추상 클래스 ElectricityData 생성
1.ElectricityData 클래스는 생성자를 통해 usage_data와 total_usage를 초기화해야 합니다.
2.usage_data는 전력 사용량 데이터를 담은 리스트로, 각 항목은 날짜와 사용량을 포함한 딕셔너리 형태입니다.
3.usage_data와 total_usage를 getter와 setter로 관리하여 캡슐화를 구현합니다.

메서드 제작
1. 추상 메서드
calculate_total_usage(): 전력 사용량 데이터를 기반으로 총 사용량을 계산
get_usage_on_date(date): 특정 날짜의 전력 사용량을 반환
2. 일반 메서드:
add_usage(date, usage): 새로운 날짜의 전력 사용량을 추가
remove_usage(date): 특정 날짜의 전력 사용량 데이터를 삭제
```

실습. 클래스 종합 프로그래밍

자식 클래스 HomeElectricityData 생성
1.부모 클래스 ElectricityData를 상속받습니다.
2.부모 클래스의 추상 메서드인 calculate_total_usage와 get_usage_on_date를 반드시 구현합니다.
3.클래스 메서드를 사용하여 electricity_usage 리스트에서 특정 날짜 범위 내의 데이터를 필터링하는 기능을 추가합니다.
4.정적 메서드를 사용하여 전력 사용량 데이터에서 가장 높은 사용량을 찾는 기능을 추가합니다.

아래내용을 구현하세요
1.데이터를 입력하여 총 전력 사용량을 계산하고 특정 날짜의 사용량을 조회할 수 있어야 합니다.
2.데이터 추가 및 삭제가 가능해야 합니다.
3.정적 메서드를 통해 가장 높은 전력 사용량과 해당 날짜를 찾으세요.
4.클래스 메서드를 통해 특정 날짜 범위 내의 사용량을 출력하세요.

출력예시

```
총 전력 사용량: 66.4
2024-11-03의 사용량: 10.8
2024-11-06 추가 후 총 전력 사용량: 82.8
특정 날짜 범위 내 사용량: [{'date': '2024-11-02', 'usage': 15.3}, {'date': '2024-11-03', 'usage': 10.8}, {'date': '2024-11-04', 'usage': 14.2}]
가장 높은 사용량: {'date': '2024-11-06', 'usage': 16.4}
```

복.습.철.저

수고하셨습니다