Day-9 Object Oriented Programming (OOP) in Python2

오늘 할 것들

- 생성자와 소멸자
- 메서드 타입 (Class / Instance / Static)
- 추상 클래스와 ABC
- 팩토리 패턴

생성자와 소멸자

Python에서 생성자(Constructor)는 객체가 생성될 때 자동으로 호출되는 특별한 메서드입니다.

일반적으로 `__init__()` 메서드를 통해 구현되며, 객체의 초기 상태를 설정하는 데 사용됩니다. 반면 소멸자(Destructor)는 `__del()__` 메서드로 구현되며, 객체가 메모리에서 제거되기 직전에 호출됩니다.

```
class MyClass:
    def __init__(self, value):  # 생성자
        print("객체가 생성됩니다!")
        self.value = value

    def __del__(self):  # 소멸자
        print("객체가 소멸됩니다!")

obj = MyClass(10)  # "객체가 생성됩니다!" 출력
# 프로그램 종료 시 또는 객체가 더 이상 참조되지 않을 때
# "객체가 소멸됩니다!" 출력
```

파이썬에서는 가비지 컬렉터가 자동으로 메모리를 관리하기 때문에, 소멸자를 명시적으로 사용하는 경우는 매우 드뭅니다.

여기까지만 보면 `__init()__` 자체가 생성자 메서드로 보이는 것 같습니다. (즉, 이 함수를 통해 직접적인 메모리 할당이 일어나는 듯 합니다)

그런데 사실 그렇지는 않습니다. 메모리를 할당하는 역할을 하는 것은 `__new()__` 이기 때문입니다. 파이썬에서 객체를 생성해보는 과정으로 알아보겠습니다.

객체가 생성될 때, 파이썬은 먼저 `__new()__` 메서드를 호출하여 객체의 메모리를 할당하고, 그 다음 `__init()__` 메서드를 호출하여 객체를 초기화합니다. 이는 다음 예제를 통해 확인할 수 있습니다:

```
class MyClass:
    def __new__(cls, *args, **kwargs):
        print("1. __new__() 호출: 메모리 할당")
        instance = super().__new__(cls)
        return instance

    def __init__(self, value):
        print("2. __init__() 호출: 객체 초기화")
        self.value = value

obj = MyClass(10)  # 두 메시지가 순서대로 출력됨
```

메서드 타입 (Class / Instance / Static)

지난 시간에는 편의를 위해 별 언급을 하지 않았습니다.

(모든 메소드를 인스턴스 메소드로 처리했습니다)

파이썬에서, 클래스의 메소드는 아래처럼 크게 세 타입으로 나눌 수 있습니다.

클래스 메서드 (Class Method)	클래스에 속한 함수로, 클래스 변수를 사용할 수 잇다. 첫 번째 인자로 cls를 사용한다. (즉, 인스턴스 없이 사용할 수 있다)
인스턴스 메서드 (Instance Method)	인스턴스에 속한 함수로, 인스턴스 변수를 사용할 수 있다. 첫번째 인자로 self를 사용한다.
스태틱 메서드 (Static Method)	클래스와 인스턴스 모두에서 호출할 수 있는 메서드로, 클래스와 인스턴스 정보에 접근할 수 없다.

실제 쓰여지는 방법도 달라집니다.

```
class MyClass:
    class_var = 0

    def __init__(self):
        self.instance_var = 1

    # 일반적인 메서드로, 객체의 인스턴스를 사용해 호출됩니다
    def instance_method(self):
        print(f"my var is {self.instance_var}")

    # cls를 인자로 받습니다. 클래스 자체에 접근할때 사용되며, 클래스 전체에 영향을 줍니
    # 언제 사용될까요?
    @classmethod
    def class_method(cls):
        print(f"my class var is {cls.class_var}")

    # 클래스에 접근하지도 않고, 인스턴스 변수에도 접근하지 않을때 사용됩니다.
    # @staticmethod 라는 데코레이터를 달고 있습니다 (self를 인자로 받지 않아도 됩니
    # 주로 유틸성 메소드를 같이 포함시킬때 사용됩니다
    @staticmethod
    def static_method():
        print("I AM STATIC METHOD")

my_class = MyClass()
my_class.instance_method() # my var is 1
MyClass.class_method() # my class var is 0
MyClass.static_method() # I AM STATIC METHOD
```

클래스메서드 좀 더 알아보기

class method를 사용하는 예제 코드를 보며, 사용방법을 좀 더 알아가보겠습니다.

클래스가 몇번 생성되었는지에 대한 카운터를 구현해보겠습니다.

```
class Counter:
    count = 0  # 클래스 변수로 카운터 초기화

    def __init__(self):
        Counter.count += 1  # 인스턴스가 생성될 때마다 카운트 증가

    @classmethod
    def get_count(cls):
        return cls.count  # 현재까지 생성된 인스턴스 수 반환

# 사용 예시
counter1 = Counter()
print(Counter.get_count())  # 1 출력
counter2 = Counter()
print(Counter.get_count())  # 2 출력
```

이 예제에서 class method를 사용하여 클래스 변수에 접근하고, 인스턴스가 생성될 때마다 카운트를 증가시키는 것을 볼 수 있습니다. 클래스 메서드는 인스턴스 생성 없이도 호출할 수 있어 이러한 용도로 매우 유용합니다.

또한 다른 사용방법도 있습니다. 클래스 메서드 안에서, 현재 클래스의 인스턴스를 만들수도 있습니다. `cls`는 클래스를 의미하므로, `cls()` 는 `__init__()` 을 호출하는 것과도 같습니다.

```
class Date:
    def __init__(self, year, month, day):
        self.year = year
        self.month = month
        self.day = day

    # 클래스 메서드를 통한 대체 생성자 패턴
    @classmethod
    def from_string(cls, date_string):
        year, month, day = map(int, date_string.split('-'))
        return cls(year, month, day)

    @classmethod
    def today(cls):
```

```
        import datetime
        today = datetime.datetime.now()
        return cls(today.year, today.month, today.day)

# 다양한 방법으로 Date 객체 생성
date1 = Date(2025, 3, 9)  # 기본 생성자
date2 = Date.from_string('2025-03-09')  # 문자열로부터 생성
date3 = Date.today()  # 현재 날짜로 생성
```

위 예제에서 볼 수 있듯이, 클래스 메서드는 주로 대체 생성자(Alternative Constructor) 패턴을 구현할 때 사용됩니다. 이를 통해 객체를 생성하는 다양한 방법을 제공할 수 있습니다.

추상 클래스와 ABC

추상 클래스(Abstract Class)는 하나 이상의 추상 메서드를 포함하는 클래스를 의미합니다. 추상 메서드는 선언만 되어 있고 구현은 되어 있지 않은 메서드로, 이를 상속받는 자식 클래스에서 반드시 구현해야 합니다. Python에서는 'abc' (Abstract Base Classes) 모듈을 통해 추상 클래스를 구현할 수 있습니다.

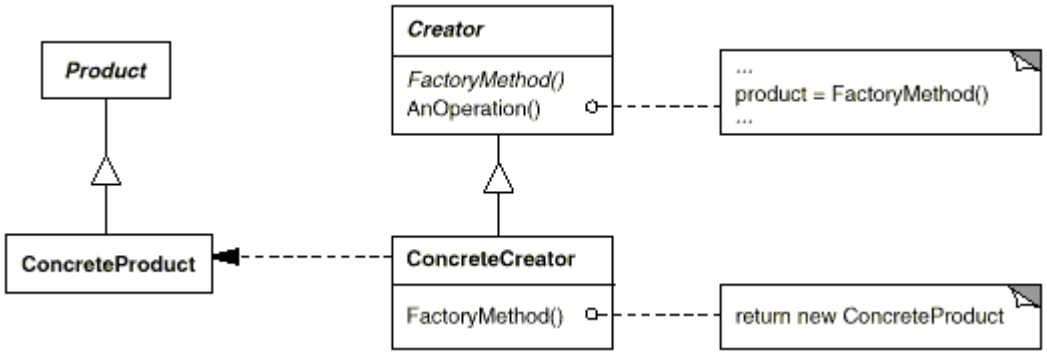
예를 들어, 동물이라는 추상 클래스가 있다면, 모든 동물은 '소리를 낸다'는 공통적인 특성을 가질 수 있지만, 구체적으로 어떤 소리를 내는지는 각 동물 클래스마다 다르게 구현되어야 합니다. Python의 ABC를 사용하면 이러한 설계를 강제할 수 있습니다.

```
from abc import ABC, abstractmethod

class Animal(ABC):
    @abstractmethod
    def make_sound(self):
        pass  # 추상 메서드는 구현하지 않음

class Dog(Animal):
    def make_sound(self):
        return "멍멍!"

class Cat(Animal):
    def make_sound(self):
        return "야옹!"
```

```
class Rat(Animal):
    # make_sound를 구현하지 않은것에 주의해주세요
        def some_sound(self):
                return "찍찍!"


dog= Dog()
cat= Cat()
rat= Rat() # 에러 발생
---
Traceback (most recent call last):
  File "/home/dongwoo/workspace/study-python/Lectures/Day-9/sample/abst
    rat= Rat()
TypeError: Can't instantiate abstract class Rat with abstract method make_sou
```

팩터리 메서드

위에 나온 내용을 바탕으로, class method와 abstract method를 이용해 자주 쓰이는 디자인 패턴을 구현해보겠습니다.



팩토리 메소드 패턴이란, **product의 세부 종류인 concreteProduct 객체의 생성을 creator의 자식 클래스인 concreteCreator에서 담당하는 것**입니다.

다음 요소로 구성됩니다.

- 객체 생성 인터페이스(creator)
- 생성할 객체 종류(concrete product)

- 선정 및 실제 생성을 담당하는 자식 클래스(concrete creator)

이때 concrete creator마다 다른 조합(종류)의 concrete product를 생성하고 보유한다. 즉, 서로 다른 product 클래스별로 각각에 대응하는 creator가 존재합니다.

자식 팩토리클래스(concrete creator)에서는, if-else 분기 등을 이용하여 인스턴스를 생성할 상품클래스(product)의 세부 상품(concrete product)를 결정한다. 부모 팩토리클래스의 메소드를 자식클래스에서 오버라이딩하여 구현한다.

예제를 통해 팩토리 메서드 패턴을 이해해보겠습니다. 다음은 간단한 로깅 시스템을 구현하는 예제입니다. (여기선 Creator를 따로 분리하지 않았습니다)

```
from abc import ABC, abstractmethod

class Logger(ABC):
    @abstractmethod
    def log(self, message):
        pass

class ConsoleLogger(Logger):
    def log(self, message):
        print(f"Console: {message}")

class FileLogger(Logger):
    def log(self, message):
        print(f"File: {message}")

class LoggerFactory:
    @classmethod
    def create_logger(cls, logger_type):
        if logger_type == "console":
            return ConsoleLogger()
        elif logger_type == "file":
            return FileLogger()
        raise ValueError("Invalid logger type")
```

이 예제에서 Logger는 product, ConsoleLogger와 FileLogger는 concrete product, LoggerFactory는 creator 역할을 합니다. LoggerFactory에서 logger_type에 따라 적

절한 Logger 객체를 생성하여 반환합니다.

이제 Creator를 별도로 분리해보겠습니다.

```
from abc import ABC, abstractmethod

# Product interface
class Product(ABC):
    @abstractmethod
    def operation(self):
        pass

# Concrete Products
class ConcreteProductA(Product):
    def operation(self):
        return "Product A operation"

class ConcreteProductB(Product):
    def operation(self):
        return "Product B operation"

# Creator (Factory) interface
class Creator(ABC):
    @abstractmethod
    def factory_method(self):
        pass

    def some_operation(self):
            # 일반적으로 비즈니스로직이 들어갑니다
        product = self.factory_method()
        return product.operation()

# Concrete Creators
class ConcreteCreatorA(Creator):
    def factory_method(self):
        return ConcreteProductA()

class ConcreteCreatorB(Creator):
```

```
    def factory_method(self):
        return ConcreteProductB()
```

이 예제에서는 Creator를 추상 클래스로 정의하고, 각각의 ConcreteCreator가 자신만의 Product를 생성하도록 구현했습니다. 이렇게 하면 새로운 제품이 추가될 때마다 기존 코드를 수정하지 않고 새로운 Creator와 Product를 추가하기만 하면 됩니다.

이렇게 하면 어떤 장점이 있을까요?

객체 생성을 서브 클래스에게 위임하여, 클래스의 유연성과 확장성을 높일 수 있습니다. 즉, 새로운 객체를 추가하거나 기존 객체를 변경해도 클래스 구조를 변경하지 않고, 팩토리 메서드만 변경하면 됩니다.

객체 생성 과정을 캡슐화하여, 객체 생성 코드의 중복을 방지하고, 유지보수성을 높일 수 있습니다.

객체 생성 과정이 복잡하거나 다양한 조건에 따라 객체를 생성해야 할 때, 팩토리 메서드 패턴을 사용하면 객체 생성 과정을 캡슐화할 수 있습니다.