

수업 내용 정리 보고서

작성일: 2026년 6월 25일

챕터 구성

1. Python Object Fundamentals and Lifecycle
2. Data Encapsulation and Memory Optimization
3. Advanced OOP: Inheritance, Polymorphism, and Method Types
4. Implementation Checklist and Common Pitfalls

파이썬 객체 기초와 생명 주기 (Python Object Fundamentals and Lifecycle)

파이썬 프로그래밍의 핵심은 객체(Object)를 설계하고 관리하는 것입니다. 본 장에서는 객체가 메모리 상에서 어떻게 탄생하고 소멸하는지, 그리고 파이썬 인터프리터가 객체를 제어하기 위해 사용하는 특별한 메커니즘에 대해 다룹니다.

1. 클래스와 인스턴스의 관계

객체지향 프로그래밍(OOP)에서 **클래스(Class)**는 객체를 만들기 위한 '설계도' 또는 '틀'을 의미하며, **인스턴스(Instance)**는 그 설계를 바탕으로 메모리에 실제로 구현된 '실체'를 의미합니다. 예를 들어, '붕어빵 틀'이 클래스라면, 그 틀에서 구워져 나온 '실제 붕어빵'이 인스턴스입니다. `a = A()` 라는 코드는 A라는 설계를 사용하여 메모리에 실제 객체 `a`를 생성하는 과정입니다.

2. 매직 메서드(Magic Method)와 던더(Dunder)

파이썬에는 양옆에 언더바 두 개(Double Under score)가 붙은 특별한 메서드들이 존재합니다. 이를 **매직 메서드(Magic Method)** 또는 **던더(Dunder)** 변수라고 부릅니다. 사용자가 직접 호출하기보다는, 파이썬 인터프리터가 특정 상황(객체 생성, 연산, 출력 등)에서 자동으로 호출하도록 예약된 시스템 함수입니다.

- `__init__(self, ...)` (생성자):** 객체가 인스턴스화될 때 자동으로 호출됩니다. 객체의 초기 상태를 설정하는 데 사용됩니다.
- `__del__(self)` (소멸자):** 객체의 참조 횟수가 0이 되어 메모리에서 해제될 때 호출됩니다. 객체의 생명 주기가 끝나는 시점에 필요한 정리 작업을 수행합니다.

3. 객체의 생명 주기와 메모리 관리

파이썬은 **참조 횟수 계산(Reference Counting)** 방식을 통해 메모리를 관리합니다. 객체를 가리키는 변수가 사라지거나 `del` 키워드를 사용하여 참조를 끊으면, 해당 객체의 참조 횟수가 감소합니다. 횟수가 0이 되는 즉시 가비지 컬렉터(Garbage Collector)가 작동하여 소멸자(`__del__`)를 호출하고 메모리를 회수합니다.

4. 모듈의 실행 제어: `if __name__ == '__main__':`

파이썬 파일은 직접 실행될 수도 있고, 다른 파일에 의해 **import(부품으로 호출)**될 수도 있습니다. 이때 `if __name__ == '__main__':` 구문은 일종의 **방화벽(Firewall)** 역할을 합니다. 이 조건문 내의 코드는 해당 파일이 '직접 실행'될 때만 작동하며, 다른 모듈에서 이 파일을 불러올(import) 때는 실행되지 않도록 격리하여 원치 않는 코드의 자동 실행을 방지합니다.

학습 확인 질문:

- 객체가 생성될 때 자동으로 실행되는 메서드의 이름은 무엇인가?
- 참조 횟수가 0이 되었을 때 호출되는 메서드는 무엇이며, 어떤 역할을 하는가?
- `if __name__ == '__main__':`을 사용하는 목적은 무엇인가?

데이터 캡슐화와 메모리 최적화 (Data Encapsulation and Memory Optimization)

효율적이고 안전한 프로그램을 만들기 위해서는 객체 내부의 데이터를 보호(캡슐화)하고, 대규모 객체 생성 시 발생하는 메모리 낭비를 최소화해야 합니다.

1. 데이터 캡슐화 (Encapsulation)

캡슐화는 객체의 내부 상태를 외부로부터 숨겨 데이터의 무결성을 유지하는 기법입니다. 파이썬에서는 변수명 앞에 언더바를 붙여 접근 권한을 암시합니다.

- `_variable` (Protected):** 내부적으로만 사용할 변수임을 나타내며, 외부 접근을 지양합니다.
- `__variable` (Private):** 클래스 외부에서 직접 접근하는 것을 더 강력하게 제한합니다.

2. 파이썬다운 Getter와 Setter: `@property`

자바(Java)와 같은 언어에서는 `get_name()`, `set_name()`과 같은 메서드를 명시적으로 호출하지만, 파이썬은 **`@property` 데코레이터**를 사용하여 변수처럼 직관적으로 접근하면서도 내부적으로는 로직을 실행할 수 있는 방식을 권장합니다.

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

    @property
    def name(self):
        """Getter: 변수처럼 접근 시 호출됨"""
        return self._name

    @name.setter
    def name(self, value):
        """Setter: 대입 연산자(=) 사용 시 호출됨 (검증 로직 포함 가능)"""
        if not value:
            raise ValueError("이름은 빈 문자열일 수 없습니다.")
        self._name = value
```

이 포맷을 사용하는 이유:

- 데이터 무결성:** Setter 내부에서 조건문을 통해 잘못된 값이 들어오는 것을 필터링할 수 있습니다.
- 하위 호환성:** 처음에는 일반 변수로 설계했다가 나중에 검증 로직이 필요해져도, 외부 코드를 수정할 필요 없이 클래스 내부만 Getter/Setter로 바꿀 수 있습니다.
- Pythonic Code:** 코드가 장황해지지 않고 깔끔하게 유지됩니다.

3. 메모리 효율성: 클래스 변수 vs 인스턴스 변수

메모리 사용 효율을 높이려면 변수의 성격을 정확히 구분해야 합니다.

구분	선언 위치	메모리 특징	공유 여부
인스턴스 변수	<code>__init__</code> 내부 (<code>self.var</code>)	객체 생성 시마다 개별 공간 할당	객체마다 독립적임
클래스 변수 (Static)	클래스 바로 아래	프로그램 초기화 시 클래스당 1개 생성	모든 인스턴스가 공유함

4. 고급 최적화: `__slots__`

기본적으로 파이썬 객체는 실시간으로 새로운 속성을 추가할 수 있도록 내부적으로 `__dict__`라는 딕셔너리를 사용합니다. 하지만 이는 메모리를 많이 소모합니다. 수만 개의 객체를 생성해야 하는 경우, `__slots__`를 사용하여 사용할 변수를 미리 고정하면 `__dict__` 공간이 사라져 메모리 사용량을 획기적으로 줄일 수 있습니다. 다만, 이 경우 지정된 변수 외의 새로운 속성을 동적으로 추가하는 것은 불가능해집니다.

심화 객체지향: 상속, 다형성, 그리고 메서드의 종류 (Advanced OOP)

객체지향의 진정한 강력함은 기존 코드를 재사용하고, 인터페이스를 통해 다양한 객체를 다루는 능력에서 나옵니다.

1. 상속(Inheritance)과 다형성(Polymorphism)

상속은 기존 클래스(부모)의 기능을 그대로 물려받아 새로운 클래스(자식)를 만드는 것입니다. 이때 `super()`를 사용하면 부모 클래스의 생성자나 메서드를 호출하여 기능을 확장할 수 있습니다.

다형성은 상속받은 메서드를 자식 클래스에서 목적에 맞게 재정의하는 **메서드 오버라이딩(Method Overriding)**을 통해 구현됩니다. 예를 들어, '스캐너'라는 부모 클래스의 `scan()` 메서드를 '스텔스 스캐너' 자식 클래스에서 우회 기술을 포함하도록 재정의하면, 동일한 이름의 함수로 서로 다른 동작을 수행하게 할 수 있습니다.

2. 파이썬 메서드의 3가지 유형 비교

메서드는 호출 방식과 전달받는 인자에 따라 다음과 같이 세 가지로 엄격히 구분됩니다.

메서드 종류	데코레이터	첫 번째 인자	주요 용도
인스턴스 메서드	(없음)	<code>self</code> (객체 자신)	개별 객체의 상태(인스턴스 변수)를 읽거나 수정
클래스 메서드	<code>@classmethod</code>	<code>cls</code> (클래스 자신)	클래스 변수 제어 또는 객체 생성 통로(팩토리) 제공
정적 메서드	<code>@staticmethod</code>	(없음)	클래스와 논리적 연관은 있으나 내부 데이터를 쓰지 않는 유틸리티 기능

실무 포인트: 클래스 내부의 상태를 변경해야 한다면 `self`나 `cls`를 사용해야 하며, 아무런 상태도 건드리지 않는 순수 함수 형태라면 `@staticmethod`를 사용하는 것이 메모리와 논리적 설계 측면에서 바람직합니다.

구현 체크리스트 및 흔한 실수 (Implementation Checklist and Common Pitfalls)

코드를 작성하거나 리뷰할 때 반드시 확인해야 할 핵심 사항들을 정리합니다.

1. 개발 시 자주 발생하는 오류 (Common Pitfalls)

- **self 누락:** 인스턴스 메서드를 정의할 때 첫 번째 파라미터로 `self`를 명시하지 않으면, 호출 시 인자가 자동으로 전달되어 에러가 발생합니다.
- **인스턴스 변수 vs 클래스 변수 혼동:** 모든 객체가 공유해야 하는 값은 클래스 바로 아래에, 객체마다 달라야 하는 값은 `__init__` 내부의 `self`. 변수명으로 선언해야 합니다.
- **상속 시 super() 누락:** 자식 클래스에서 `__init__`을 새로 정의할 때 `super().__init__()`을 호출하지 않으면 부모 클래스에서 정의한 변수들을 사용할 수 없게 됩니다.
- **Getter/Setter 호출 오류:** `@property`를 사용한 경우, 호출할 때 `p.name()`처럼 괄호를 붙이면 안 됩니다. 변수처럼 `p.name`으로 접근해야 합니다.

2. 최종 구현 체크리스트

1. [] 클래스 변수와 인스턴스 변수가 의도에 맞게 분리되었는가?
2. [] 데이터 보호가 필요한 변수에 `Getter/Setter(@property)`가 적용되었는가?
3. [] 대규모 객체 생성이 예상되는 경우 `__slots__`를 통한 메모리 최적화를 고려하였는가?
4. [] 모듈 실행 제어를 위해 `if __name__ == '__main__':` 블록을 사용하였는가?
5. [] 상속 관계에서 부모 클래스의 초기화 로직이 `super()`를 통해 호출되는가?