PyThing

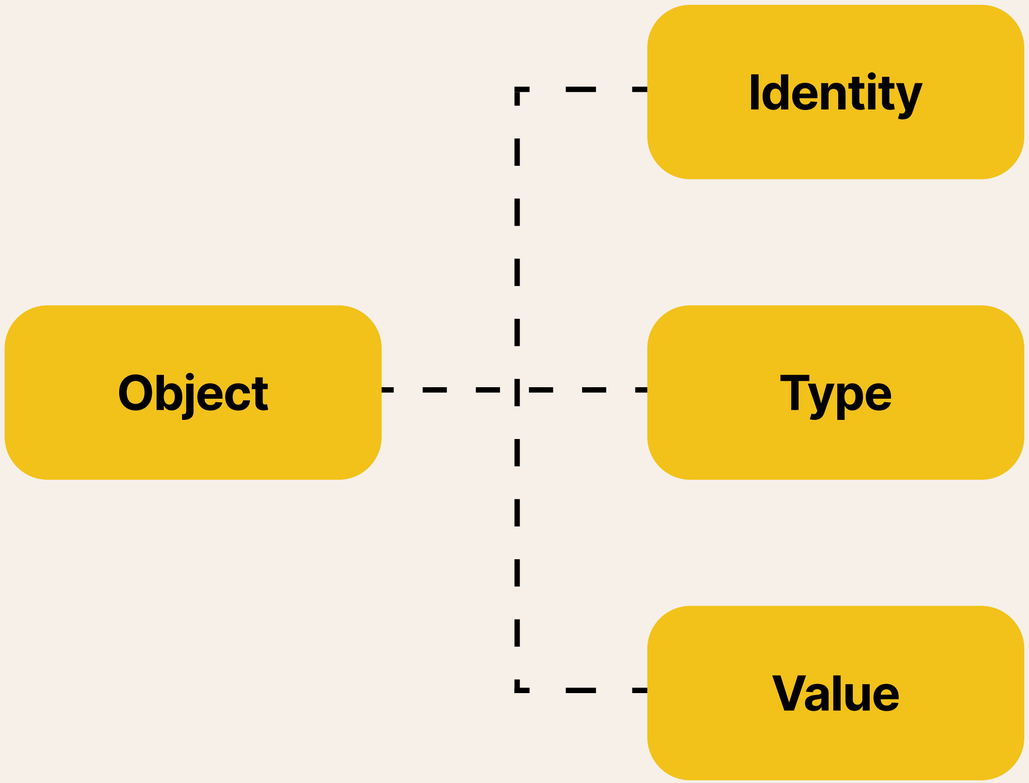
OOP in Python

PyThing 이승헌

T.O.C

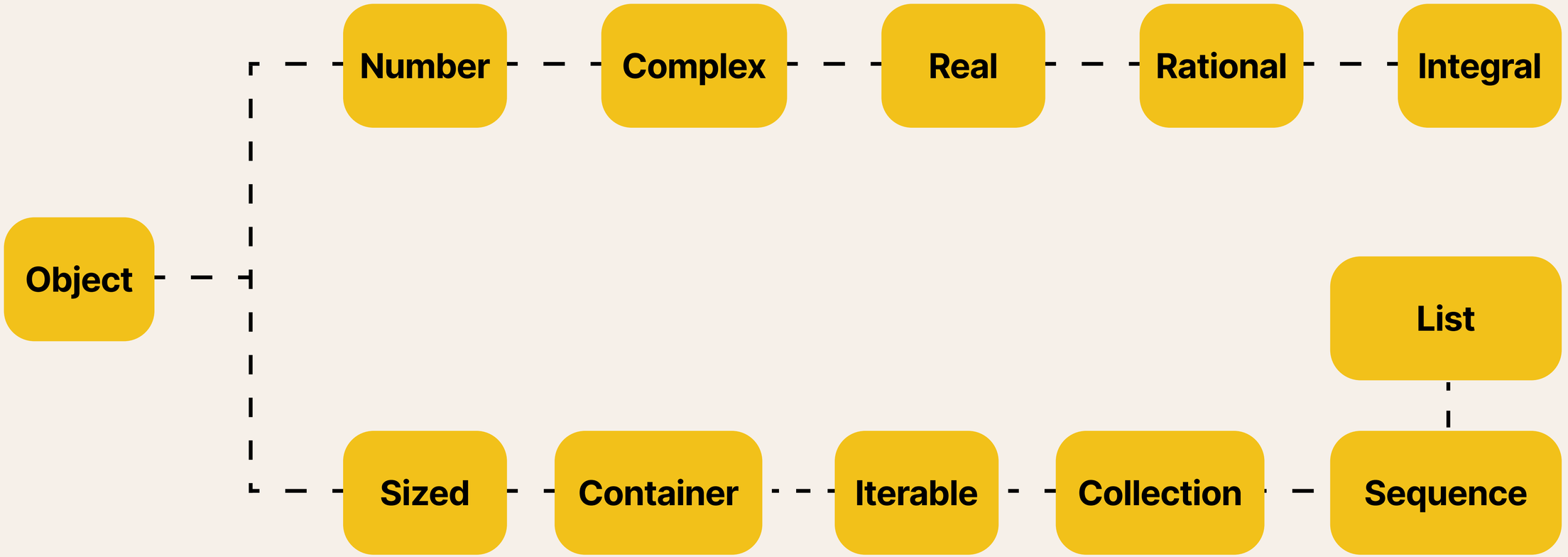
Object Oriented Programming in Python

The Whole Thing is Object!




```
# in REPL


>>> x = [1,2,3]  # object
>>> id(x)
4342794560       # Identity
>>> type(x)
<class 'list'>   # Type
>>> x
[1, 2, 3]        # Value


```

The Whole Thing is Object!



Special Methods

```python
class Car:
    def __init__(self, engine):
        self.engine = engine

    def __repr__(self):
        return f"Car(engine={self.engine})"

```

Python’s approach to operator overloading, **allowing classes to define their own behavior** with respect to language operators.

 # dunder method # magic method

Special Methods

__new__, __init__, __del__, __repr__, __str__, __bytes__, __format__, __hash__,
__bool__, __getattr__, __getattribute__, __setattr__, __delattr__, __dir__,
__eq__, __ne__, __lt__, __le__, __gt__, __ge__, __get__, __set__, __delete__,
__set_name__, __len__, __length_hint__, __getitem__, __setitem__,
__delitem__, __missing__, __iter__, __reversed__, __contains__, __call__,
__enter__, __exit__, __neg__, __pos__, __abs__, __invert__, __add__, __sub__,
__mul__, __matmul__, __truediv__, __floordiv__, __mod__, __divmod__,
__pow__, **__lshift__**, **__rshift__**, __and__, __xor__, __or__, __radd__, __rsub__,
__rmul__, __rmatmul__, __rtruediv__, __rfloordiv__, __rmod__, __rdivmod__,
__rpow__, __rlshift__, __rrshift__, __rand__, __rxor__, __ror__, __iadd__,
__isub__, __imul__, __imatmul__, __itruediv__, __ifloordiv__, __imod__,
__ipow__, __ilshift__, __irshift__, __iand__, __ixor__, __ior__, __complex__,
__int__, __float__, __index__, __round__, __trunc__, __floor__, __ceil__,
__class_getitem__, __instancecheck__, __subclasscheck__,
__init_subclass__, __prepare__, __class__, __reduce__, __reduce_ex__,
__getstate__, __setstate__

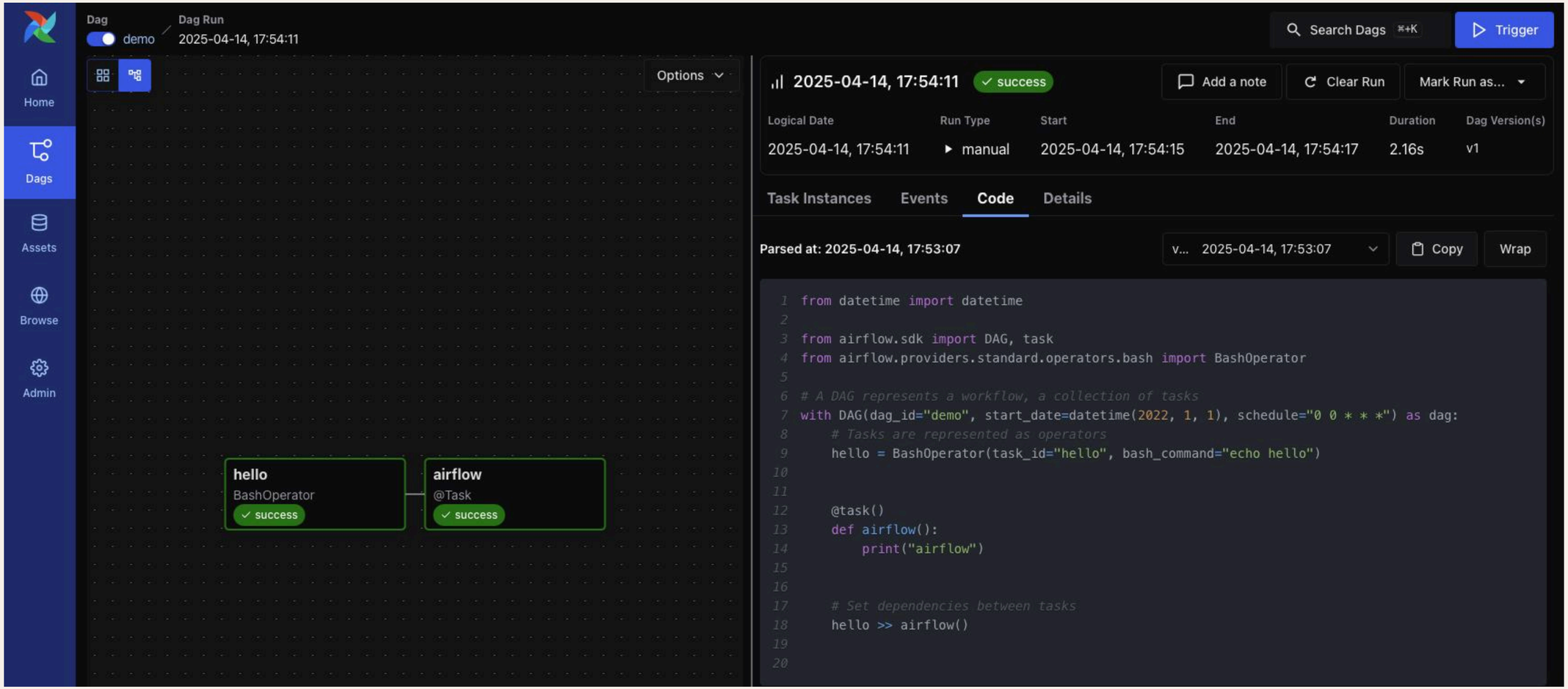
109개의 Special Methods

Use Case: Apache Airflow

Apache Airflow® is an open-source platform for developing, scheduling, and monitoring batch-oriented workflows.

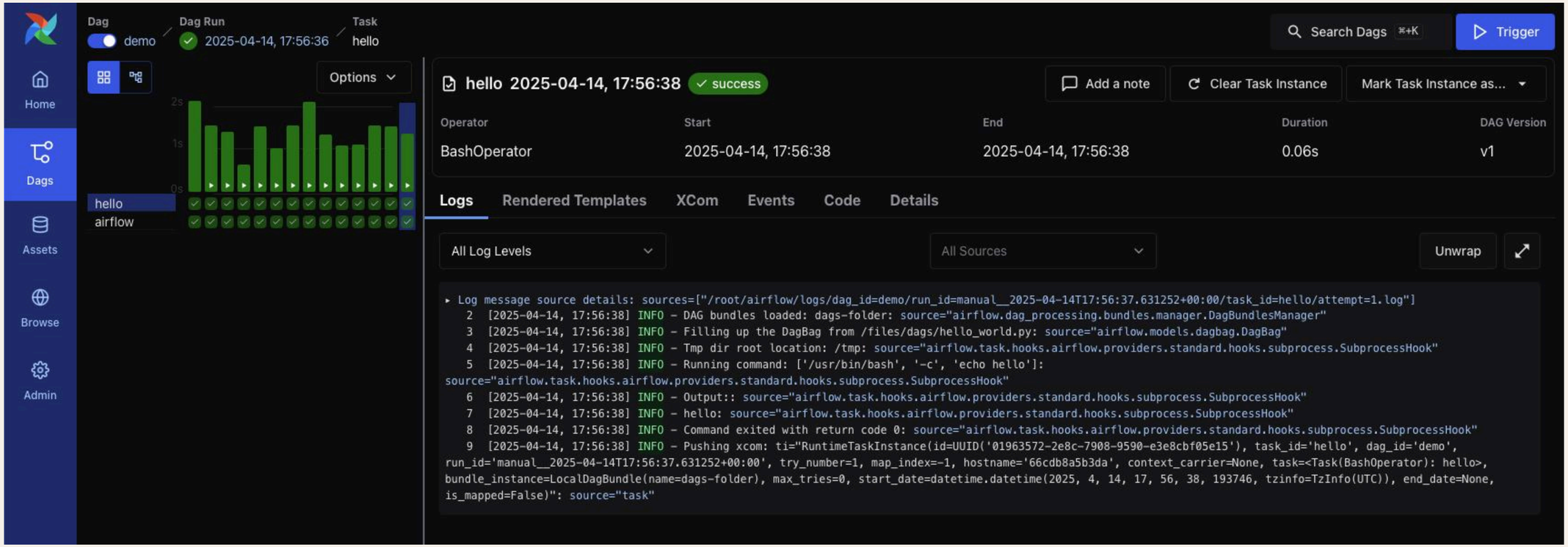
 # Data Engineering # ETL # Machine Learning Pipeline # Python

Use Case: Apache Airflow



각 **Operator** 간의 의존성을 **DAG(Directed Acyclic Graph)**로 표현

Use Case: Apache Airflow



실행 성공 여부와 로그 확인

Use Case: Apache Airflow

```python
from datetime import datetime

from airflow.sdk import DAG, task
from airflow.providers.standard.operators.bash import BashOperator

# A DAG represents a workflow, a collection of tasks
with DAG(dag_id="demo", start_date=datetime(2022, 1, 1), schedule="0 0 * * *") as dag:
    # Tasks are represented as operators
    hello = BashOperator(task_id="hello", bash_command="echo hello")

    @task()
    def airflow():
        print("airflow")

    # Set dependencies between tasks
    hello >> airflow()
```

8 >> 3

```python
# Set dependencies between tasks
hello >> airflow()
```

```python
class DependencyMixin:
    """Mixing implementing common dependency setting methods like >> and <<."""
    # ...
    def __lshift__(self, other: DependencyMixin | Sequence[DependencyMixin]):
        """Implement Task << Task."""
        self.set_upstream(other)
        return other

    def __rshift__(self, other: DependencyMixin | Sequence[DependencyMixin]):
        """Implement Task >> Task."""
        self.set_downstream(other)
        return other
```

airflow/task-sdk/src/airflow/sdk/definitions/_internal/mixins.py

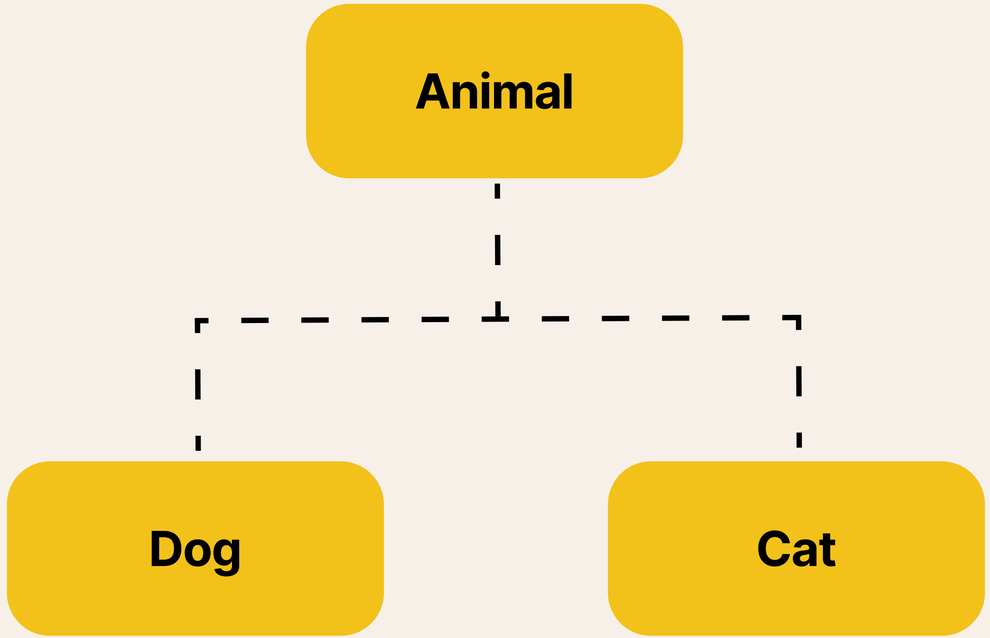
Object Oriented Programming in Python

Object Relationships

1. **Inheritance**
2. Composition
3. Association
4. Aggregation

Inheritance

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

    def speak(self):
        return "소리를 낸다"

# 자식 클래스 (서브 클래스)
class Dog(Animal):
    def speak(self):
        return f"{self.name}가 말한다: 멍멍!"

class Cat(Animal):
    def speak(self):
        return f"{self.name}가 말한다: 야옹!"

# 사용 예
dog = Dog("초코")
cat = Cat("나비")

print(dog.speak())  # 초코가 말한다: 멍멍!
print(cat.speak())  # 나비가 말한다: 야옹!
```



Inheritance

abc — Abstract Base Classes

Source code: Lib/abc.py

```python
from abc import ABC, abstractmethod

class Animal(ABC):
    @abstractmethod
    def sound(self):
        pass

class Dog(Animal):
    def sound(self):
        return "멍멍"

```

Nominal Subtyping

Protocols

(Originally specified in **PEP 544**.)

```python
from typing import Protocol

class Speaker(Protocol):
    def speak(self) -> str:
        ...

class Dog:
    def speak(self) -> str:
        return "멍멍"

class Cat:
    def speak(self) -> str:
        return "야옹"

def make_it_talk(animal: Speaker):
    print(animal.speak())

```

Structural Subtyping

abc.ABC

```python
from abc import ABC, abstractmethod

class Animal(ABC):
    @abstractmethod
    def sound(self):
        pass

class Dog(Animal):
    def sound(self):
        return "멍멍"


if __name__ == '__main__':
    dog = Dog()
    print(dog.sound())

```

- **Nominal Subtyping**
  - 이름 기반 타입 관계 판단
  - 타입 간 상속 관계 명시적 선언
- **ABC**
  - Abstract Base Classes를 정의하는 모듈
- **abstractmethod**
  - abstract method를 정의할 때 사용
  - 반드시 하위 클래스에서 override
- **runtime 검증**
  - __abstractmethods__을 통해 검증

typing.Protocol

```python
from typing import Protocol

class Speaker(Protocol):
    def speak(self) -> str:
        ...

class Dog:
    def speak(self) -> str:
        return "멍멍"

class Cat:
    def speak(self) -> str:
        return "야옹"

def make_it_talk(animal: Speaker):
    print(animal.speak())

if __name__ == '__main__':
    dog = Dog()
    make_it_talk(dog)

```

- **Structural Subtyping**
  - 구조 기반 타입 관계 판단
  - 타입 간 상속 관계 암묵적 선언
- **typing.Protocol**
  - Structural Subtyping을 위한 인터페이스 선언 형식
- **정적분석기를 통한 검증**
  - mypy, pyright를 통한 정적 분석

abc.ABC vs. typing.Protocol

abc.ABC	vs.	typing.Protocol
명시적 상속	기본 철학	구조적 서브타이핑
반드시 상속해야 함	상속 필요 여부	상속하지 않아도 됨
구현 강제, 인터페이스 계약	목적	타입 검사 시 구조 검증
추상 메서드 강제, 인스턴스 생성 차단	런타임 효과	기본적으로는 타입 검사용 (런타임 무관)

E.O.D