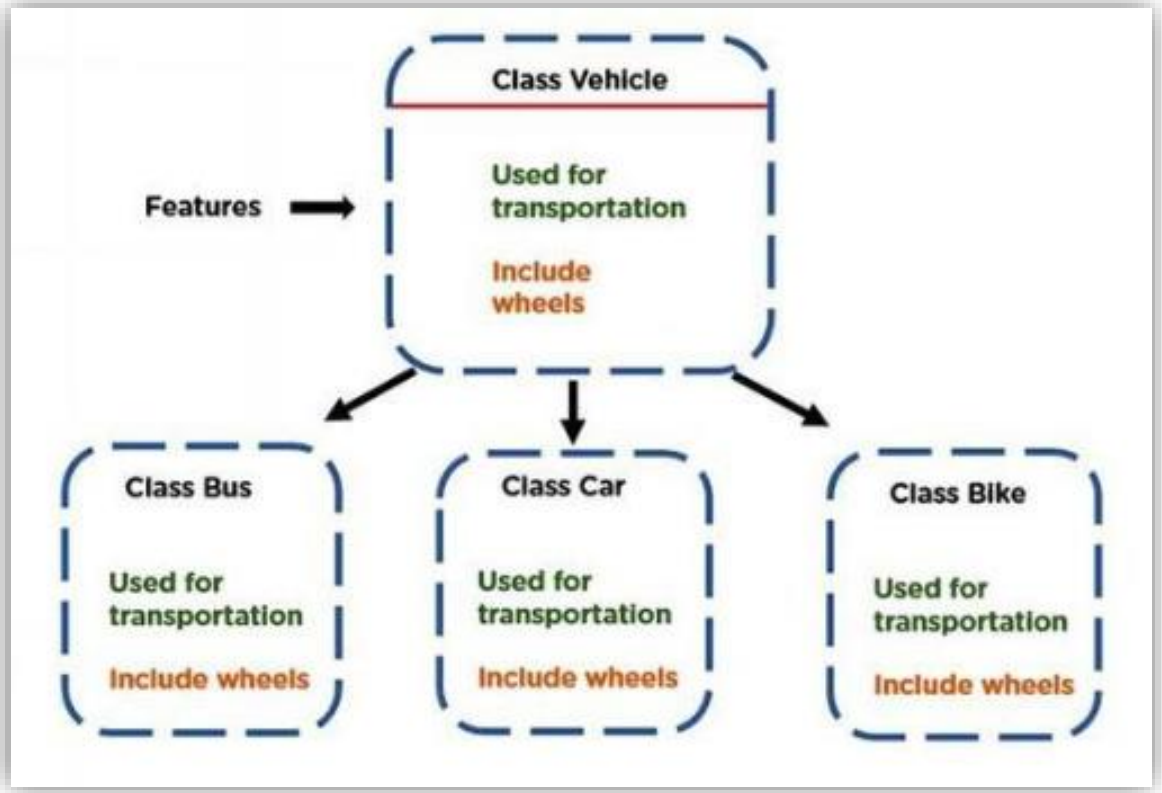
Python.OOP

Python.OOP

OOP - Object-Oriented Programming

- 컴퓨터 프로그래밍의 패러다임 중 하나
- 컴퓨터 프로그램을 명령어의 목록으로 보는 시각에서 벗어나 여러 개의 독립된 단위, 즉 "객체"들의 모임으로 파악하고자 하는 것.
- 각각의 객체는 메시지를 주고받고, 데이터를 처리할 수 있다 -wiki



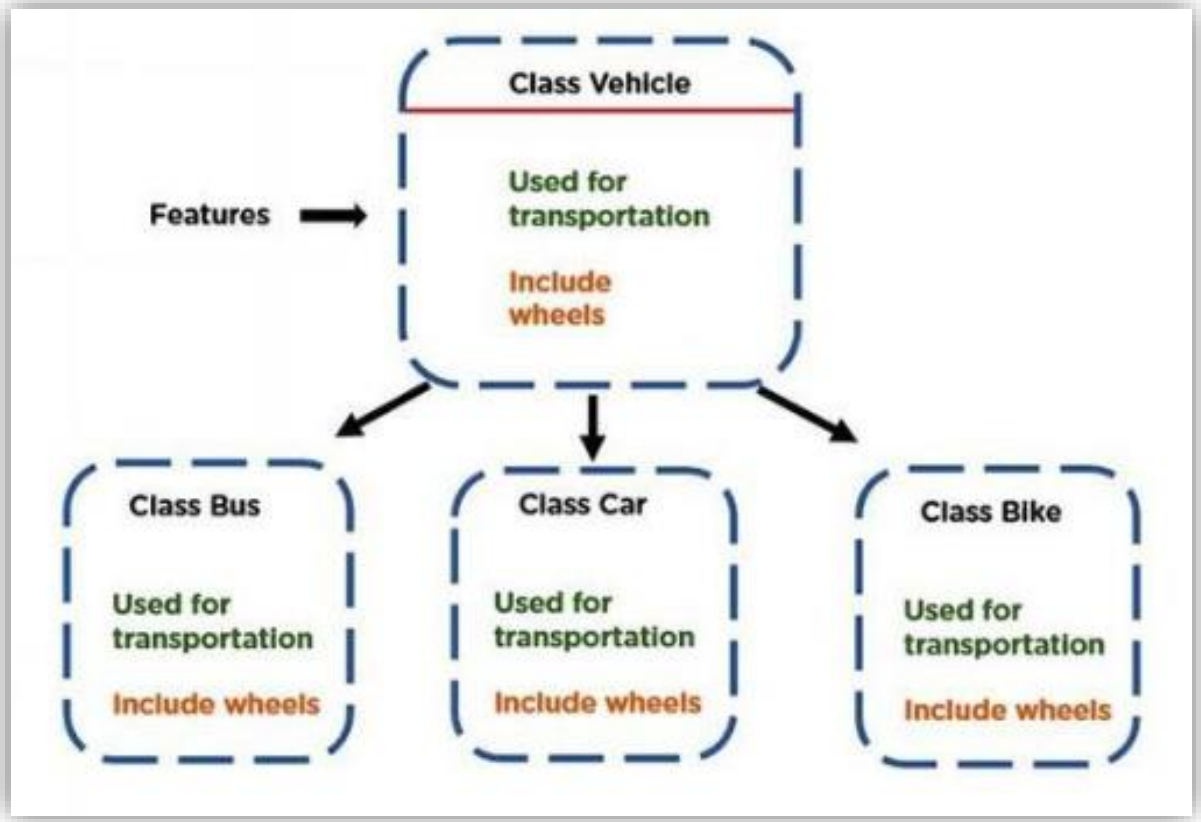
Python.OOP

OOP - Object-Oriented Programming

- 실습

  1.03.OOP.ipynb

  TASK : Sub Class 정의



Python.OOP

OOP *vs* Procedural Programming (절차적 프로그래밍)

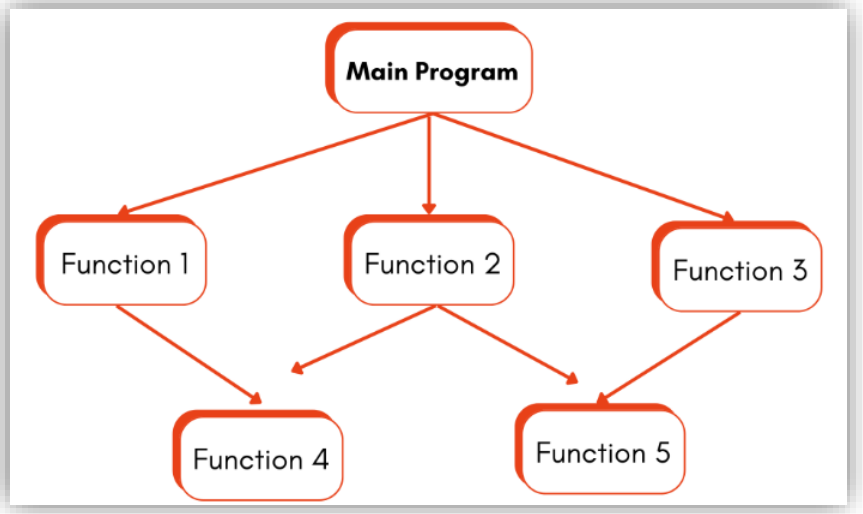
- **Procedural Programming (절차적 프로그래밍)**

  ✓ 절차지향 프로그래밍 혹은 절차지향적 프로그래밍 프로그래밍 패러다임의 일종

  ✓ 때때로 명령형 프로그래밍과 동의어로 쓰이기도

  ✓ 프로시저 호출의 개념을 바탕으로 하고 있는 프로그래밍 패러다임을 의미하기도 - wiki

  * 프로시져 : 함수, 메소드, 루틴, 서브루틴 등



https://reliasoftware.com/blog/difference-between-procedural-and-object-oriented-programming

Python.OOP

OOP *vs* Procedural Programming (절차적 프로그래밍)

- **Procedural Programming (절차적 프로그래밍) :**

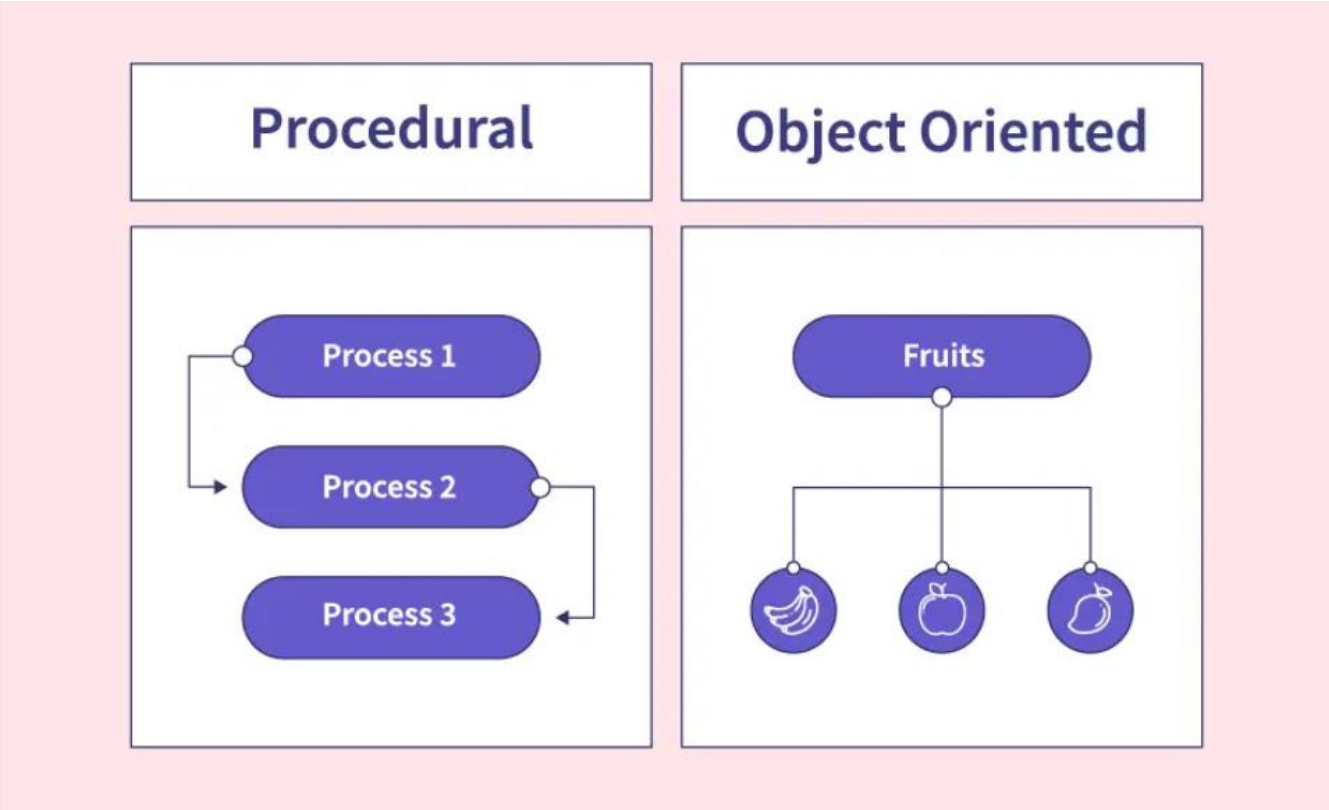
  : 프로그램을 **일련의 절차(순서)와 함수의 흐름**으로 구성하는 방식

  - ✓ **절차지향 프로그래밍 혹은 절차지향적 프로그래밍 프로그래밍 패러다임의 일종**
  - ✓ **때때로 명령형 프로그래밍과 동의어로 쓰이기도**
  - ✓ **프로시저 호출의 개념을 바탕으로 하고 있는 프로그래밍 패러다임을 의미하기도 - wiki**

*** 프로시져 : 함수, 메소드, 루틴, 서브루틴 등**

Python.OOP

OOP *vs* Procedural Programming (절차적 프로그래밍)



Python.OOP

OOP *vs* Procedural Programming (절차적 프로그래밍)

- **실습**

  1.03.OOP.v.PP.ipynb

  직사각형의 넓이와 둘레 계산

  TASK : 두 코드의 차이점 파악

Python.OOP

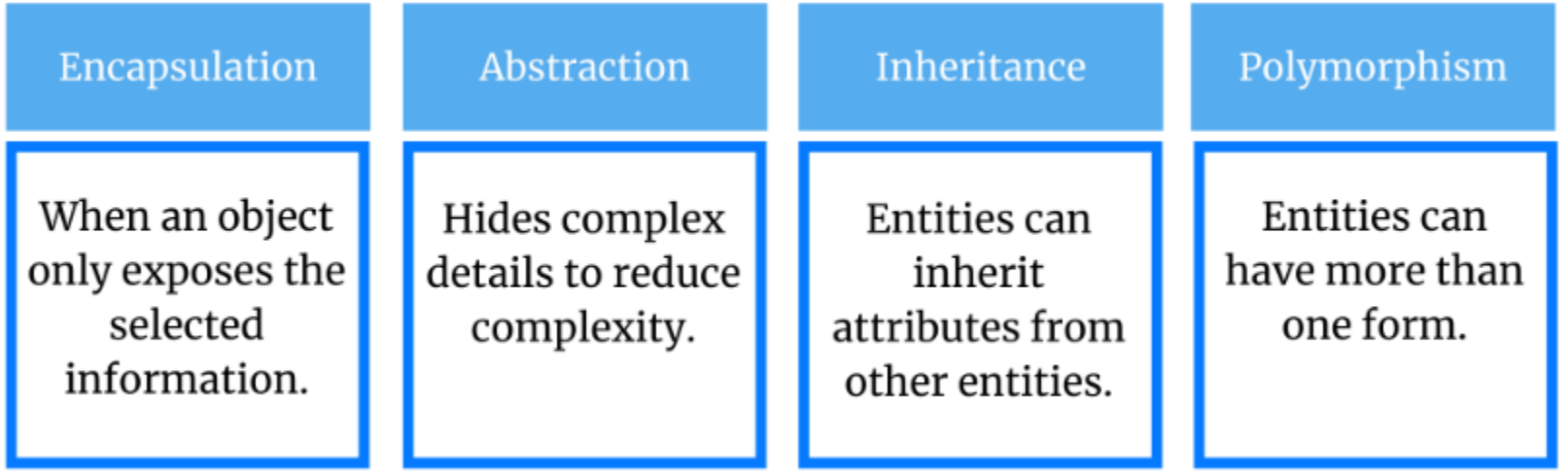
OOP *vs* Procedural Programming (절차적 프로그래밍)

	절차지향 프로그래밍 (Procedural)	객체지향 프로그래밍 (OOP)
특징	함수(Function) 기반	객체(Object) 기반
데이터 관리	변수와 함수가 따로 존재	객체 내부에서 속성으로 관리
코드 재사용성	낮음 (반복 코드 발생)	높음 (클래스를 사용하여 객체 생성)
유지보수	데이터와 로직이 분리 -> 복잡	데이터와 로직이 함께 관리 ->유지보수 용이
확장성	새로운 기능 추가 시 많은 코드 변경	상속(Inheritance) 등을 통해 확장 수월

Python.OOP

OOP - 특징

- **캡슐화(Encapsulation)** – 정보 은닉 및 접근 제한자
- **상속(Inheritance)** – 코드 재사용성과 확장성 증가
- **다형성(Polymorphism)** – 동일 인터페이스, 다른 동작 구현
- **추상화(Abstraction)** – 핵심 정보만 노출하여 복잡성 감소



Python.OOP

OOP - 특징

▪ **캡슐화(Encapsulation)** – 정보 은닉 및 접근 제한자

✓ 객체 내부의 데이터(속성)를 보호하고, 외부에서 직접 접근할 수 없도록 제한
-> 데이터 무결성 유지, 불필요한 접근 방지

```
class BankAccount:
    def __init__(self, balance):
        self.__balance = balance  # Private 변수 (외부에서 직접 접근 불가)

    def deposit(self, amount):
        self.__balance += amount

    def get_balance(self):  # Getter 메서드
        return self.__balance

account = BankAccount(1000)
account.deposit(500)
print(account.get_balance())  # 출력: 1500
# print(account.__balance)    # 오류 발생 (접근 불가)
```

변수 이름 앞에 __(언더스코어 두 개)가 붙여 비공개(Private) 변수로 정의
-> 외부에서 직접 접근 불가능

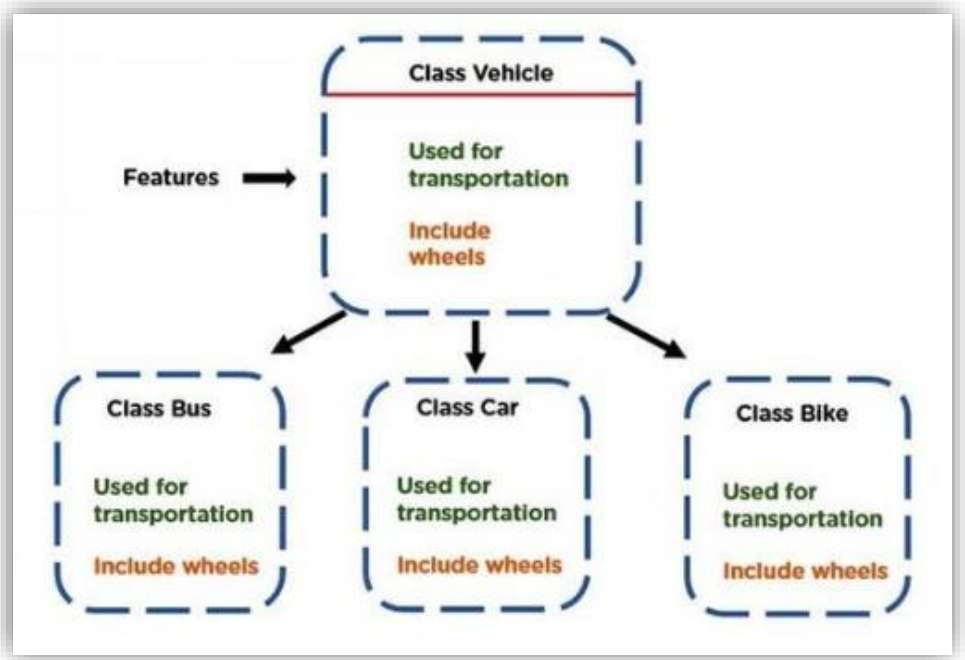
Python.OOP

OOP - 특징

- **상속(Inheritance)** – 코드 재사용성과 확장성 증가

✓ 부모 클래스(상위 클래스)의 속성과 메서드를 자식 클래스(하위 클래스)가 물려받아 사용
-> 코드 재사용성을 높이고, 유지보수 용이

1.03.OOP.ipynb



Python.OOP

OOP - 특징

▪**다형성(Polymorphism)** – 동일 인터페이스, 다른 동작 구현

✓ 같은 메서드(함수)가 다양한 객체에서 다르게 동작

```
class BankAccount:
    def __init__(self, balance):
        self.__balance = balance  # Private 변수 (외부에서 직접 접근 불가)

    def deposit(self, amount):
        self.__balance += amount

    def get_balance(self):  # Getter 메서드
        return self.__balance

account = BankAccount(1000)
account.deposit(500)
print(account.get_balance())  # 출력: 1500
# print(account.__balance)    # 오류 발생 (접근 불가)
```

Python.OOP

OOP - 특징

▪ **추상화(Abstraction)** – 핵심 정보만 노출하여 복잡성 감소

✓ 사용자가 필요한 기능만 사용할 수 있도록 불필요한 내부 구현을 숨김
✓ 복잡한 시스템을 간단한 인터페이스로 제공
-> 코드의 가독성과 유지보수성 Up

```
from abc import ABC, abstractmethod

class Vehicle(ABC):  # 추상 클래스
    @abstractmethod    # @abstractmethod를 붙여서 추상 메서드로 지정
    def start_engine(self):
        pass

class Car(Vehicle):
    def start_engine(self):
        return "자동차 엔진이 켜졌습니다."

class Motorcycle(Vehicle):
    def start_engine(self):
        return "오토바이 엔진이 켜졌습니다."

car = Car()
print(car.start_engine())  # 출력: 자동차 엔진이 켜졌습니다.
```

ABC를 상속받아 추상 클래스(Abstract Class) 생성
직접 인스턴스를 생성 X
상속받아 구현

THANK YOU