Day-8 Object Oriented Programming (OOP) in Python

개요

프로그래밍 공부를 하다보면 항상 나오는 이야기가 있습니다. **객체지향**을 잘 지키냐, **객체지향**에서의 의존성 역전이... 객체지향 프로그래밍은 매우 일반적으로 사용되는 프로그램 설계 방법론입니다. 영어로는 Object Oriented Programming이라고 하고, 줄여서 OOP라고 하기도 합니다.

OOP는 서로 연관된 데이터들과 행위들을 한 객체에 그룹화 할 수 있게 해줍니다. 일종의 청사진격인 class를 정의하고, 그것들로부터 여러 object들을 만들어 낼 수 있습니다. OOP는 이를 통해 실제 세계의 개념을 프로그램에 녹여낼 수 있게 도와줍니다. 그리고 재사용성을 높여주며, 확장성을 높여줍니다.

오늘은 파이썬에서의 객체지향 프로그래밍을 알아볼 것입니다.

객체지향? 절차지향?

절차지향 프로그래밍?

객체 지향의 프로그래밍이 어땠을지 생각해봅시다.

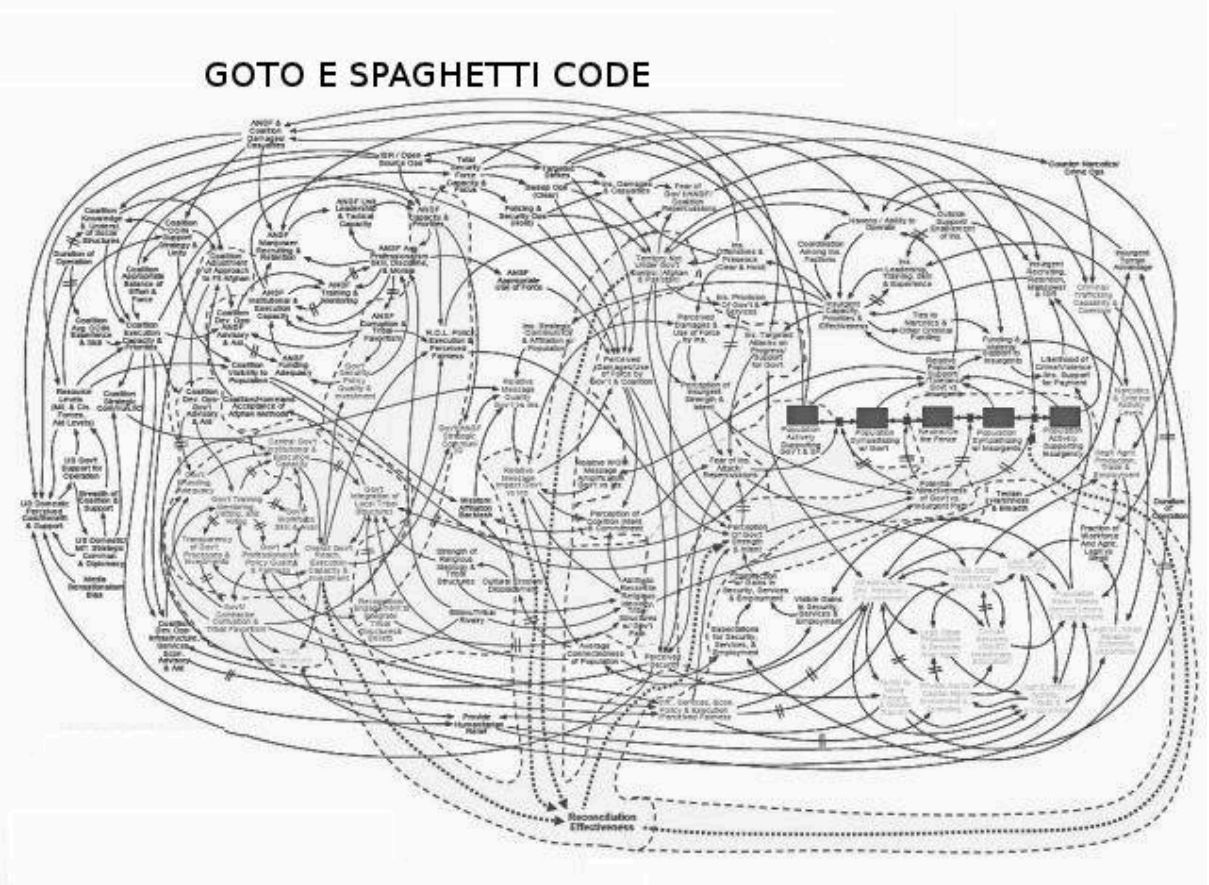
우리가 처음 프로그래밍을 시작할 때 우리는 대부분 순차적 처리를 중요시합니다. 이런 순차적 처리를 중심으로 하는 프로그래밍 밥법론을 **절차지향 프로그래밍**이라고 부릅니다.

이런 방법론에는 몇 가지 특징이 존재합니다.

- 하나의 큰 기능을 처리하기 위해 작은 단위의 기능들로 나누어 처리하는 Top-Down 접근 방식으로 설계됩니다.
- 데이터와 함수를 별개로 취급합니다.
- 컴퓨터의 처리구조와 유사해 실행 속도가 빠릅니다.
- 프로그램이 커질수록 구조가 복잡해져 유지보수가 어렵습니다.

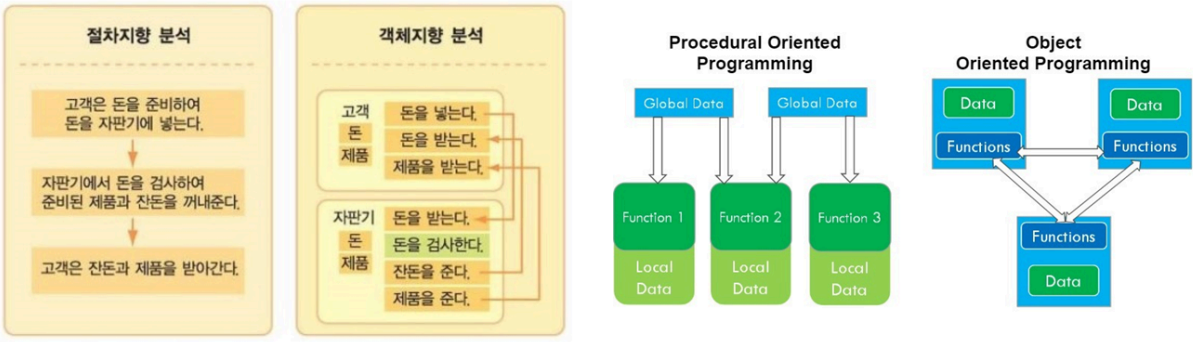
대표적인 절차지향 언어에는 **C언어, Visual Basic, Fortran, Pascal** 등이 있습니다.

(물론 엄밀히 따지면, 꼭 그렇지도 않습니다. 리눅스는 C언어로 개발됐지만, OOP로 개발되었기 때문입니다.)



그런데 이런 개발방식에는 문제가 있습니다. 프로젝트 규모가 커지면 코드가 복잡해지기 쉽습니다. (보통 스파게티코드 라고 부릅니다) 그리고 이런 특성때문에 유지보수가어려워집니다.

객체지향 프로그래밍 (OOP)



이런 단점 때문에 OOP에서는 각 성격의 속성(데이터)과 절차들을 하나의 덩어리로 묶어 생각합니다. 예를 들어 `사람` 을 나타내는 객체에는 이름, 나이, 주소와 같은 **속성(property)** 이 있을 수 있습니다. 그리고 걷기, 숨쉬기, 뛰기 같은 **행위(behaviors)** 가 있을 수 있습니다. 이 행위를 일반적으로 **메서드(method)**라고 합니다.

또 다른 예시로 `email` 과 관련된 객체를 구현해야 한다고 생각해봅시다. 그러면 다음과 같은 속성과 행위들을 email이라는 객체와 함께 묶을 수 있을 것입니다.

속성

- 수신자 리스트
- 참조 리스트
- email 내용
- 첨부파일

행위

- 메일 보내기
- 첨부파일 첨부하기

복잡성이 증가하는 요즘 프로그래밍 환경에서, OOP는 사실상 표준이 되었습니다.

이런 객체지향 프로그래밍은 일반적으로 4가지 특성으로 설명됩니다.

1. **캡슐화 (Encapsulation) :**
    - 객체의 속성과 행위를 하나로 묶는 것을 의미합니다.
    - 각 속성에 접근하고 변경 하는 메서드를 별도로 정의함으로서, 모듈간의 결합도를 떨어뜨려 유지보수성을 높입니다.
    - 일반적으로 많은 언어들은 변경 가능성이 있는 세부 구현 내용을 private 으로 설정할 수 있게 합니다. (이를 통해 다른 프로그래머가 시스템을 망치는 것을 막을 수 있습니다)

2. **상속 (Inheritance)** :
    - 상속은 자식 클래스가 부모 클래스의 속성과 메서드를 이어받을 수 있도록 해줍니다.
    - 이를 통해 클래스 간에 계층적 관계를 만들 수 있게 해줍니다.
    - 이런 특성은 코드의 재사용성을 높이고, 중복 구현을 방지하게 해줍니다.
3. **추상화 (Abstraction)** :
    - 데이터나 프로세스 등을 의미가 비슷한 개념이나, 의미 있는 표현으로 정의하는 과정을 이야기합니다.
    - 추상화의 목적은 클래스 구현과 세부 동작을 분리하는 것입니다.
    - 이를 통해 클래스간 상호작용을 간단하게 만듭니다. 개발자는 각 객체가 '어떻게' 동작하는지보다 '무엇'을 하는지에 집중할 수 있습니다.
4. **다형성 (Polymorphism)** :
    - 동일한 인터페이스를 만들어, 서로 다른 유형의 객체를 동일한 기본 유형의 인스턴스로 처리할 수 있게 해줍니다.
    - 파이썬은 `Duck typing`을 지원하는데, 덕분에 실제 클래스에 대해 걱정할 필요 없이 객체의 속성과 메서드에 액세스 할 수 있게 해줍니다
        - Duck typing이 없는 언어는 각 객체의 타입을 검사 후, 내부 메서드를 사용할 수 있습니다.
        - 그러나 파이썬은 타입을 검사하지 않고, 다만 호출 시점에 특정 메서드가 구현되어 있지 않으면 런타임 에러를 내뱉습니다.

절차지향 프로그래밍에서, 객체는 그저 데이터 덩어리를 가리킵니다. 그러나 객체지향 프로그래밍에서, 객체는 이런 특성들로 실제 프로그램의 구조를 나타냅니다.

파이썬에서 Class 정의해보기

클래스 및 인스턴스 속성

파이썬에서, `class` 키워드로 class를 정의할 수 있습니다. 그리고 `__init__()` 으로 클래스의 각 인스턴스가 어떤 속성을 가져야 할지 선언할 수 있습니다.

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

1. `self.name = name` 은 `name` 속성을 만들고 ,초기화 시 입력받은 `name` 인자를 할당합니다.
2. `self.age = age`는 `age` 속성을 만들고, 초기화시 입력받은 `age` 인자를 할당합니다.

반면 **클래스 속성**은 모든 클래스 인스턴스에 대해 동일한 값을 갖는 속성입니다. 외부의 변수 이름에 값을 할당해 정의할 수 있습니다.

```
class Employee:
    company = "my-company" # 추가
  def __init__(self, name, age):
    self.name = name
    self.age = age
```

이제 이걸 **인스턴스화** 해봅시다.

```
class Employee:
    company = "my-company"
  def __init__(self, name, age):
    self.name = name
    self.age = age

ken = Employee("ken", 23)
print(ken.name) # ken
print(ken.age) # 23
print(ken.company) # my-company
```

인스턴스 메서드

인스턴스 메서드는 클래스 내부에서 정의하고, 해당 클래스의 인스턴스에서만 호출 할 수 있는 함수입니다. `self`를 인자로 받습니다.

```
class Employee:
    company = "my-company"
  def __init__(self, name, age):
    self.name = name
    self.age = age

  def work(self):
      print("집에 보내주세요")

    def speak(self, sound):
        return f"{self.name} says, {sound}"

ken = Employee("ken", 23)
ken.work() # 집에 보내주세요
print(ken.speak("나는 일이 좋아")) # ken says, 나는 일이 좋아
```

상속

상속은 한 클래스가 다른 클래스의 속성과 메서드를 가져오는 프로세스입니다. 새로 형성된 클래스를 **자식 클래스**라고 하고, 자식 클래스를 파생시키는 클래스를 **부모 클래스**라고 합니다.

```
class Parent:
    hair_color = "brown"

class Child(Parent):
    pass

joe = Child()
print(joe.hair_color) # brown
```

자식 클래스는 부모클래스의 메서드나 속성을 재정의 할 수도 있습니다.

```
class Parent:
    hair_color = "brown"

class Child(Parent):
    hair_color = "black"

joe = Child()
print(joe.hair_color) # black
```

다음 예제는 자식 클래스가 부모의 속성을 상속받고, 이를 확장하는 과정을 보여줍니다.

```
class Parent:
    hair_color = "brown"
    def __init__(self):
        self.language = ["English"]

class Child(Parent):
    hair_color = "black"
    def __init__(self):
        super().__init__()
        self.language.append("Korean")

joe = Child()
print(joe.hair_color) # black
print(joe.language) # ['English', 'Korean']
```

이제 부모의 메서드를 상속받고, 그것을 확장하는 것을 확인해봅시다. super를 통해 부모 클래스에 액세스 할 수 있습니다.

```
class Parent:
    hair_color = "brown"
    def __init__(self):
```

```
        self.language = ["English"]
    def says(self):
        print(f"She said, 공부해라")

class Child(Parent):
    hair_color = "black"
    def __init__(self):
        super().__init__()
        self.language.append("Korean")
    def says(self):
        print(f"He answered in {self.language[-1]}, 싫어요!")

ally = Parent()
joe = Child()
print(joe.hair_color) # black
print(joe.language) # ['English', 'Korean']

print(ally.hair_color)

ally.says()
joe.says()
```

super를 좀 더 적극적으로 사용해봅시다.

```
class Dog:
    def __init__(self, name, age):
        self.name = name
        self.age = age
    def bark(self, sound):
        return f"{self.name}({self.age}): {sound}!"

class Yorkie(Dog):
    def bark(self, sound="...zzz"):
        return super().bark(sound)
```

```
koko = Yorkie("koko",5)
print(koko.bark())
```

접근 제어하기 (property)

이제 파이썬의 property에 대해 알아볼 것입니다. 다음 코드를 보고 문제점을 확인해봅시다.

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age
    def is_enlisted_in_military(self):
        return (self.age > 20)

jongho = Person("종호", 23)
# ....
jongho.age = 10 # 누군가 잘못 접근했다!
# ...
print(jongho.is_enlisted_in_military()) # False, 의도한 결과가 아님
```

문제는 이 뿐만이 아닙니다. 이 경우는 접근해서는 안 될 데이터에 잘못 쓴 경우입니다.

```
# ...
suyoung = Person("수영", -28) # 음수를 잘못 썼다!
```

대부분의 언어에서는 `private`과 `public`, `protected` 키워드를 지원하는데요. 파이썬에서는 이를 명시적으로 지원하지는 않습니다. 대신 `_` 를 앞에 붙여 이것이 private 속성임을 표기합니다. (암시적 약속입니다)

Person 클래스 코드를 수정해봅시다. 변수명을 변경하고, property를 적용해 볼것입니다.

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

```
        self._age = age  # aget -> _age

    @property
    def age(self):
        print("나이를 리턴합니다.")
        return self._age

    @age.setter
    def age(self, age_value):
        if age_value < 0:
            print("나이는 0보다 작을 수 없습니다.")
            return
        if age_value < self._age:
            print("나이는 줄어들 수 없습니다.")
            return
        self._age = age_value

jongho = Person("jongho", 23)
print(jongho.age)
jongho.age = 10
print(jongho.age)
```

아래와 같은 출력이 나옵니다

```
나이를 리턴합니다.
23
나이는 줄어들 수 없습니다.
나이를 리턴합니다.
23
```

이렇게 property 데코레이터가 붙은 메서드는 특이한 방식으로 기능이 변합니다. 이제 `person.age` 는 property가 붙은 `age` 메서드를 실행시키게 됩니다. 일반적으로 property 데코레이터가 붙은 메서드는, 그 클래스의 `getter` 임을 의미합니다.

마찬가지로, `@age.setter` 데코레이터가 붙은 메서드는, `Person 클래스 인스턴스.age = 어떤값` 코드가 실행될 때, 대신 실행됩니다. 이런 기능을 프로퍼티라고 합니다.

중요한 것은 실제 클래스 인스턴스에는 `_age` 프로퍼티가 있을 뿐, `age` 변수가 있는 것이 아니라는 것입니다.

다음 코드로 그것을 확인할 수 있습니다

```
print(jongho.__dict__)
# {'name': 'jongho', '_age': 23}
```

이를 바탕으로 좀 더 그럴듯한 코드를 짜봅시다

```
class Coordinate:
    def __init__(self, lat: float, long: float) -> None:
        self._latitude = self._longitude = None
        self.latitude = lat
        self.longitude = long

    @property
    def latitude(self) -> float:
        return self._latitude

    @latitude.setter
    def latitude(self, lat_value: float) -> None:
        if -90 <= lat_value <= 90:
            self._latitude = lat_value
        else:
            raise ValueError(f"유효하지 않는 위도 값: {lat_value}")

    @property
    def longitude(self) -> float:
        return self._longitude

    @longitude.setter
    def longitude(self, long_value: float) -> None:
        if -180 <= long_value <= 180:
            self._longitude = long_value
        else:
```

```
            raise ValueError(f"유효하지 않은 경도 값: {long_value}")
    def __str__(self):
        return f"위도: {self._latitude}, 경도: {self._longitude}"


coord = Coordinate(37.5024625930912, 127.02862987558395)
print(coord) # 위도: 37.5024625930912, 경도: 127.02862987558395

cord2 = Coordinate(50.5024625930912, 230.02862987558395)
print(cord2) # 런타임 에러 발생
```

물론 자유도가 높은 파이썬답게, 모든 인스턴스 변수에 이걸 강제하는 것은 아닙니다.

데이터클래스 (PEP-557)

클래스를 사용하다보면 아래와 같은 초기화 코드를 많이 작성해야 합니다

```
class Sample():
    def __init__(self, x, y):
        self.x = x
        self.y = y
```

이런 코드는 보일러 플레이트 (boilerplate) 혹은, 보이러 플레이트 코드라고 한다. Python 3.7부터는 dataclasses 모듈의 dataclass 데코레이터, field를 이용하여 중복되는 코드를 제거하고 단순화할 수 있습니다.

```
from dataclasses import dataclass, field
import random

# isbn 생성함수
def generate_isbn():
    # 13자리 숫자 생성
    return "".join([str(random.randint(0, 9)) for _ in range(13)])
```

```
@dataclass
class Book:
    language = "Learn Err Day"  # 클래스 변수
    # writer = field(init=False)    # 이렇게 초기화 하지 않고 남길수도 있습니다
    # writer: str                   # 이러면 인스턴스 변수

    title: str  # 인스턴스 변수, 반드시 할당해줘야 합니다

    is_sale: bool = True  # 이렇게 기본 값을 정해줄 수도 있습니다


    # shared_list: list = []        # err! (mutable 한 값을 할당할 수 없습니다)

    # 때문에 아래와 같이 field를 사용해야 합니다
    unshared_list: list = field(default_factory=lambda: ["hello", "world"])

    # 다음과 같이 입력받을 값을 정할 수 도 있습니다
    isbn: str = field(init=False, default_factory=generate_isbn)

    # 기존 field들로 새로운 필드를 만들수도 있습니다
    def __post_init__(self) -> None:
        self.searchh_str = f"{self.title} - {self.isbn}"

    def __str__(self) -> str:
        return f"{self.title} - {self.isbn}"
```

위 코드를 바탕으로 인스턴스를 만들어봅시다.

```
hamlit = Book("Hamlet")
print(hamlit.__dict__)
# {'title': 'Hamlet', 'is_sale': True, 'unshared_list': ['hello', 'world'], 'isbn': '0382
```

각종 주석을 제외하고, dataclass가 아닌 버전과 비교하면 꽤 편리하다는 것을 알 수 있습니다.

```
@dataclass
class Book:
    language = "Learn Err Day"
    title: str
    is_sale: bool = True
    unshared_list: list = field(default_factory=lambda: ["hello", "world"])
    isbn: str = field(init=False, default_factory=generate_isbn)

    def __post_init__(self) -> None:
        self.searchh_str = f"{self.title} - {self.isbn}"

    def __str__(self) -> str:
        return f"{self.title} - {self.isbn}"
```

```
class Book:
    language = "Learn Err Day"

    def __init__(self, title, is_sale=True, writer=None, unshared_list=None, isbn=
        self.title = title
        self.is_sale = is_sale
        if unshared_list is None:
            shared_list = ["hello", "world"]
        self.isbn = generate_isbn()
        self.search_str = f"{self.title} - {self.isbn}"

    def __str__(self):
        return f"{self.title} - {self.isbn}"
```