Overview

Object Oriented Programming

출처: 최성철 교수

프로그램을 여럿이 개발할 경우

코드를 어떻게 작성해야 할까?

만들어 놓은 코드를

재사용하고 싶다!

프로그램을 기능별로 나누어

재사용하는 방법

① 함수 ② 객체 ③ 모듈

클래스와 객체

- 객체 지향 언어의 이해 -

[생각해보기]

수강신청 프로그램을 작성한다.

어떻게 해야할까?

[생각해보기]

① 수강신청이 시작부터 끝까지 순서대로 작성

② 수강신청 관련 주체들(교수, 학생, 관리자) 의 행동(수강신청, 과목 입력)과 데이터(수강과목, 강의 과목) 들을 중심으로 프로그램 작성 후 연결

[생각해보기]

두 가지 모두 가능

최근엔 ②번 방식이 주류

이러한 기법을

객체 지향 프로그램 이라 함

객체 지향 프로그램

- Object-Oriented Programming, OOP
- 객체: 실생활에서 일종의 물건
  속성(Attribute)와 행동(Action)을 가짐
- OOP는 이러한 객체 개념을 프로그램으로 표현
  속성은 variable, 행동은 함수로 표현됨
- 파이썬 역시 객체 지향 프로그램 언어

객체 지향 프로그램

- 인공지능 축구 프로그램을 작성한다고 가정
- 객체 종류: 팀, 선수, 심판, 공
- Action : 선수 – 공을 차다, 패스하다.
  심판 – 휘슬을 불다, 경고를 주다.
- Attribute : 선수 – 선수 이름, 포지션, 소속팀
  팀 – 팀 이름, 팀 연고지, 팀소속 선수

객체 지향 프로그래밍

- OOP는 설계도에 해당하는 클래스(class)와
  실제 구현체인 인스턴스(instance) 로 나눔

붕어빵틀
(Class)

붕어빵
(인스턴스)

객체 지향 프로그래밍

http://rupil.egloos.com/m/5697622

Objects in Python

Class 구현하기 in Python

- 축구 선수 정보를 Class로 구현하기

```
class SoccerPlayer(object):
    def __init__(self, name, position, back_number):
        self.name = name
        self.position = position
        self.back_number = back_number

    def change_back_number(self, new_number):
        print("선수의 등번호를 변경합니다 : From %d to %d" % (self.back_number, new_number))
        self.back_number = new_number
```

Class 구현하기 in Python - 선언

class 선언은

```
class SoccerPlayer(object):
```

class 예약어 class 이름 상속받는 객체명

[알아두면 상식] Naming

- 변수와 Class명 함수명은 짓는 방식이 존재
- snake_case : 띄워쓰기 부분에 "_" 를 추가
  뱀 처럼 늘여쓰기, 파이썬 함수/변수명에 사용
- CamelCase: 띄워쓰기 부분에 대문자
  낙타의 등 모양, 파이썬 Class명에 사용

https://goo.gl/sVb2yd

Class 구현하기 in Python - Attribute

Attribute 추가는 __init__ , self와 함께!
__init__은 객체 초기화 예약 함수

```
class SoccerPlayer(object):
    def __init__(self, name, position, back_number):
        self.name = name
        self.position = position
        self.back_number = back_number
```

[알아두면 상식] 파이썬에서 _ 의미

- _는 특수한 예약 함수나 변수에 사용됨

예) __main__ , __add__ , __str__

```
class SoccerPlayer(object):

    def __str__(self):
        return "Hello, My name is %s. I play in %s in center " % ₩
       (self.name, self.position)

jinhyun = SoccerPlayer("Jinhyun", "MF", 10)
print(jinhyun)
```

Class 구현하기 in Python - Function

Function(Action) 추가는 기존 함수와 같으나, 반드시 self를 추가해야만 class 함수로 인정됨

```
class SoccerPlayer(object):
    def change_back_number(self, new_number):
        print("선수의 등번호를 변경합니다 :
              From %d to %d" % ₩
             (self.back_number, new_number))
        self.back_number = new_number
```

Objects(Instance) 사용하기

Object 이름 선언과 함께 초기값 입력 하기

```
jinhyun = SoccerPlayer("Jinhyun", "MF", 10)
```

객체명 Class명 _init_ 함수 Interface, 초기값

```
def __init__(self, name, position, back_number):
```

```
class SoccerPlayer(object):
        ........

jinhyun = SoccerPlayer("Jinhyun", "MF", 10)
print("현재 선수의 등번호는 :", jinhyun.back_number)
jinhyun.change_back_number(5)
print("현재 선수의 등번호는 :", jinhyun.back_number)
```

Class 구현하기 in Python

```
class SoccerPlayer(object):
    def __init__(self, name, position, back_number):
        self.name = name
        self.position = position
        self.back_number = back_number

    def change_back_number(self, new_number):
        print("선수의 등번호를 변경합니다 : From %d to %d" % (self.back_number, new_number))
        self.back_number = new_number


jinhyun = SoccerPlayer("Jinhyun", "MF", 10)
print("현재 선수의 등번호는 :", jinhyun.back_number)
jinhyun.change_back_number(5)
print("현재 선수의 등번호는 :", jinhyun.back_number)
```

이렇게 하면

뭐가 좋나요?

5명 Soccer Player 정보 저장하기

- 이차원 리스트 사용해보기

```
names = ["Jin", "Sungchul", "Ronaldo", "Hong", "Seo"]
positions = ["MF","DF", "CF", "WF", "GK"]
numbers = [10, 15, 20, 3, 1]

players = [[name, position, number ]for name, position, number in zip(names, positions, numbers)]
print(players)
print(players[0])
```

- Class로 선언하기

```
player_objects = [SoccerPlayer(name, position, number) for name, position, number in zip(names, positions, numbers)]
print(player_objects[0])
```

구현 가능한 OOP 만들기 - 노트북

- Note를 정리하는 프로그램
- 사용자는 Note에 뭔가를 적을 수 있다.
- Note에는 Content가 있고, 내용을 제거할 수 있다.
- Note는 Notebook에 삽입된다.
- Notebook은 Note가 삽일 될 때 페이지를 생성하며, 최고 300페이지까지 저장 가능하다
- 300 페이지가 넘으면 더 이상 노트를 삽입하지 못한다.

Class

	Notebook	Note
method	add_note remove_note get_number_of_pages	write_content remove_all
variable	title page_number notes	content

```
class Note(object):
    def __init__(self, content = None):
        self.content = content

    def write_content(self, content):
        self.content = content

    def remove_all(self):
        self.content = ""

    def __str__(self):
        return self.content
```

content

write_content

remove_all

```
class NoteBook(object):
    def __init__(self, title):
        self.title = title
        self.page_number = 1
        self.notes = {}

    def add_note(self, note, page = 0):
        if self.page_number < 300:
            if page == 0:
                self.notes[self.page_number] = note
                self.page_number += 1
            else:
                self.notes = {page : note}
                self.page_number += 1
        else:
            print("Page가 모두 채워졌습니다.")

    def remove_note(self, page_number):
        if page_number in self.notes.keys():
            return self.notes.pop(page_number)
        else:
            print("해당 페이지는 존재하지 않습니다")

    def get_number_of_pages(self):
        return len(self.notes.keys())
```

title, page_number, notes

add_note

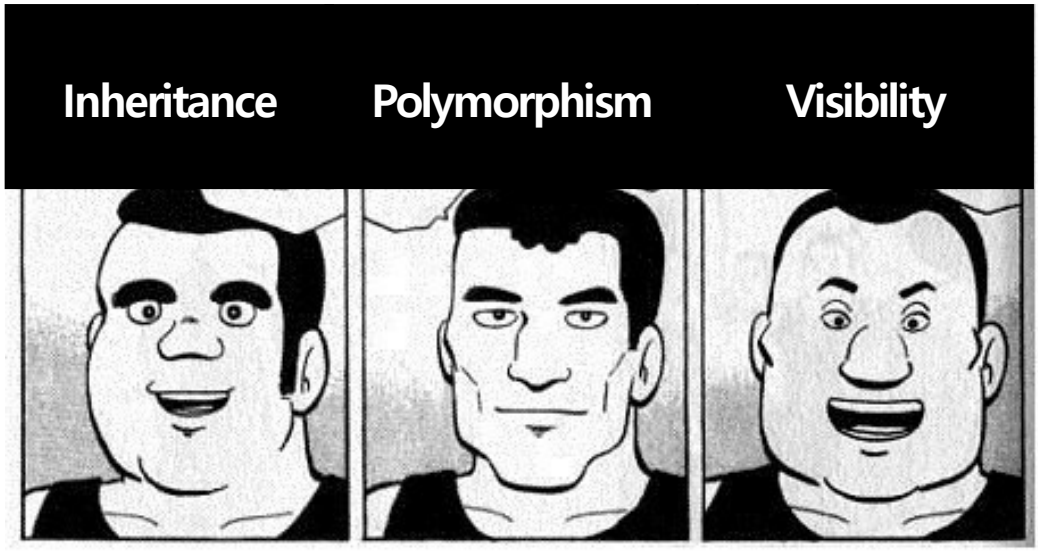
remove_note

get_number_of_pages

객체 지향 언어의 특징

실제 세상을 모델링

이를 위한 특징들



https://goo.gl/RmNwUj

오늘 어차피 모든 걸

이해 못함

Inheritance
상속

상속 (Inheritance)

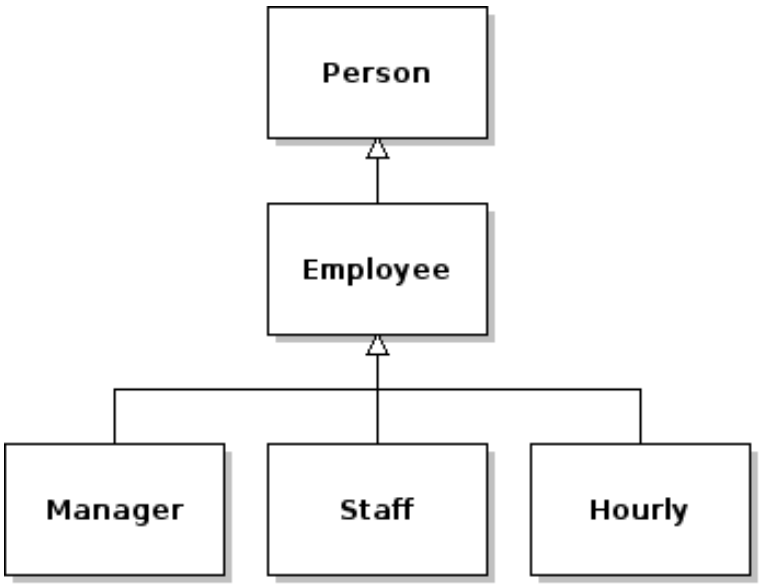
- 부모클래스로 부터 속성과 Method를 물려받은 자식 클래스를 생성 하는 것

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

class Korean(Person):
    pass

first_korean = Korean("Sungchul", 35)
print(first_korean.name)
```

OOP 특징 - 상속 (Inheritance)



OOP 특징 - 상속 예제

```
class Person(object):  # 부모 클래스 Person 선언
    def __init__(self, name, age, gender):
        self.name = name
        self.age = age
        self.gender = gender

    def about_me(self):  # Method 선언
        print("저의 이름은 ", self.name, "이구요, 제 나이는 ", str(self.age), "살 입니다.")
```

Source: http://blog.eairship.kr/m/post/286

OOP 특징 - 상속 예제

```
class Employee(Person):  # 부모 클래스 Person으로 부터 상속
    def __init__(self, name, age, gender, salary, hire_date):
        super().__init__(name, age, gender)  # 부모객체 사용
        self.salary = salary
        self.hire_date = hire_date  # 속성값 추가

    def do_work(self):  # 새로운 메서드 추가
        print("열심히 일을 합니다.")

    def about_me(self):  # 부모 클래스 함수 재정의
        super().about_me()  # 부모 클래스 함수 사용
        print("제 급여는 ", self.salary, "원 이구요, 제 입사일은 ", self.hire_date, "
입니다.")
```

Source: http://blog.eairship.kr/m/post/286

Polymorphism
다형성

다형성 (Polymorphism)

- 같은 이름 메소드의 내부 로직을 다르게 작성
- Dynamic Typing 특성으로 인해 파이썬에서는 같은 부모클래스의 상속에서 주로 발생함
- 중요한 OO의 개념 그러나 너무 깊이 알 필요X

다형성 (Polymorphism)

```
class Animal:
    def __init__(self, name):  # Constructor of the class
        self.name = name

    def talk(self):  # Abstract method, defined by convention only
        raise NotImplementedError("Subclass must implement abstract method")

class Cat(Animal):
    def talk(self):
        return 'Meow!'

class Dog(Animal):
    def talk(self):
        return 'Woof! Woof!'

animals = [Cat('Missy'),
           Cat('Mr. Mistoffelees'),
           Dog('Lassie')]

for animal in animals:
    print(animal.name + ': ' + animal.talk())
```

https://goo.gl/lBy9uf

Visibility
가시성

가시성 (Visibility)

- 객체의 정보를 볼 수 있는 레벨을 조절하는 것
- 누구나 객체 안에 모든 변수를 볼 필요가 없음

1) 객체를 사용하는 사용자가 임의로 정보 수정
2) 필요 없는 정보에는 접근 할 필요가 없음
3) 만약 제품으로 판매한다면? 소스의 보호

[알아두면 상식] Encapsulation

- 캡슐화 또는 정보 은닉 (Information Hiding)
- Class를 설계할 때, 클래스 간 간섭/정보공유의 최소화
- 심판 클래스가 축구선수 클래스 가족 정보를 알아야 하나?
- 캡슐을 던지듯, 인터페이스만 알아서 써야함

Visibility Example

- Product 객체를 Inventory 객체에 추가
- Inventory에는 오직 Product 객체만 들어감
- Inventory에 Product가 몇 개인지 확인이 필요
- Inventory에 Product items는 직접 접근이 불가

Visibility Example

```
class Product(object):
    pass
```

```
class Inventory(object):
    def __init__(self):
        self.__items = []

    def add_new_item(self, product):
        if type(product) == Product:
            self.__items.append(product)
            print("new item added")
        else:
            raise ValueError("Invalid Item")

    def get_number_of_items(self):
        return len(self.__items)
```

Visibility Example

```
my_inventory = Inventory()
my_inventory.add_new_item(Product())
my_inventory.add_new_item(Product())
print(my_inventory.get_number_of_items())

print(my_inventory.__items)
my_inventory.add_new_item(object)
```

Visibility Example

- Product 객체를 Inventory 객체에 추가
- Inventory에는 오직 Product 객체만 들어감
- Inventory에 Product가 몇 개인지 확인이 필요
- Inventory에 Product items 접근 허용

Visibility Example

```
class Inventory(object):
    def __init__(self):
        self.__items = []

    @property
    def items(self):
        return self.__items

my_inventory = Inventory()
my_inventory.add_new_item(Product())
my_inventory.add_new_item(Product())
print(my_inventory.get_number_of_items())

items = my_inventory.items
items.append(Product())
print(my_inventory.get_number_of_items())
```

Visibility

```
class Inventory(object):
    def __init__(self):
        self.__items = []
```

Private 변수로 선언 남들이 접근 못함

Property decorator 숨겨진 변수를 반환하게 해줌

```
    @property
    def items(self):
        return self.__items

my_inventory.items
```

Property decorator로 함수를 변수처럼 호출