Python Object-Oriented Programming

TEAMLAB director
최성철

boostcamp AI Tech

파이썬을 개발하는 방법

만들어 놓은 코드를

재사용하고 싶다!

클래스와 객체

- 객체 지향 언어의 이해 -

boostcamp AI Tech

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

두 가지 모두 가능

최근엔 ②번 방식이 주류

이러한 기법을

객체 지향 프로그램 이라 함

객체지향 프로그래밍 개요

- Object-Oriented Programming, OOP

- 객체: 실생활에서 일종의 물건
  속성(Attribute)와 행동(Action)을 가짐

- OOP는 이러한 객체 개념을 프로그램으로 표현
  속성은 변수(variable), 행동은 함수(method)로 표현됨

- 파이썬 역시 객체 지향 프로그램 언어

- 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

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

붕어빵틀
(Class)

붕어빵
(인스턴스)

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

직접 구현을 해봐야 알 수 있음

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 선언하기

- class 선언, object는 python3에서 자동 상속

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

class 예약어 class 이름 상속받는 객체명

[알아두면 상식] Python naming rule

- 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

http://bit.ly/3aP5Yjh

Attribute 추가하기

- 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 변수 그리고 함수명 변경(맨글링)으로 사용

예) __main__ , __add__ , __str__ , __eq__

```
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)
```

https://corikachu.github.io/articles/python/python-magic-method

method 구현하기

- method(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

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

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

객체명 Class명 _init_ 함수 Interface, 초기값

```
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)
```

OOP

Implementation Example

구현 가능한 OOP 만들기 - 노트북

- Note를 정리하는 프로그램
- 사용자는 Note에 뭔가를 적을 수 있다.
- Note에는 Content가 있고, 내용을 제거할 수 있다.
- 두 개의 노트북을 합쳐 하나로 만들 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 scheme

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

Note class


```
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 __add__(self, other):
        return self.content + other.content

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

content

write_content

remove_all

NoteBook class


```
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

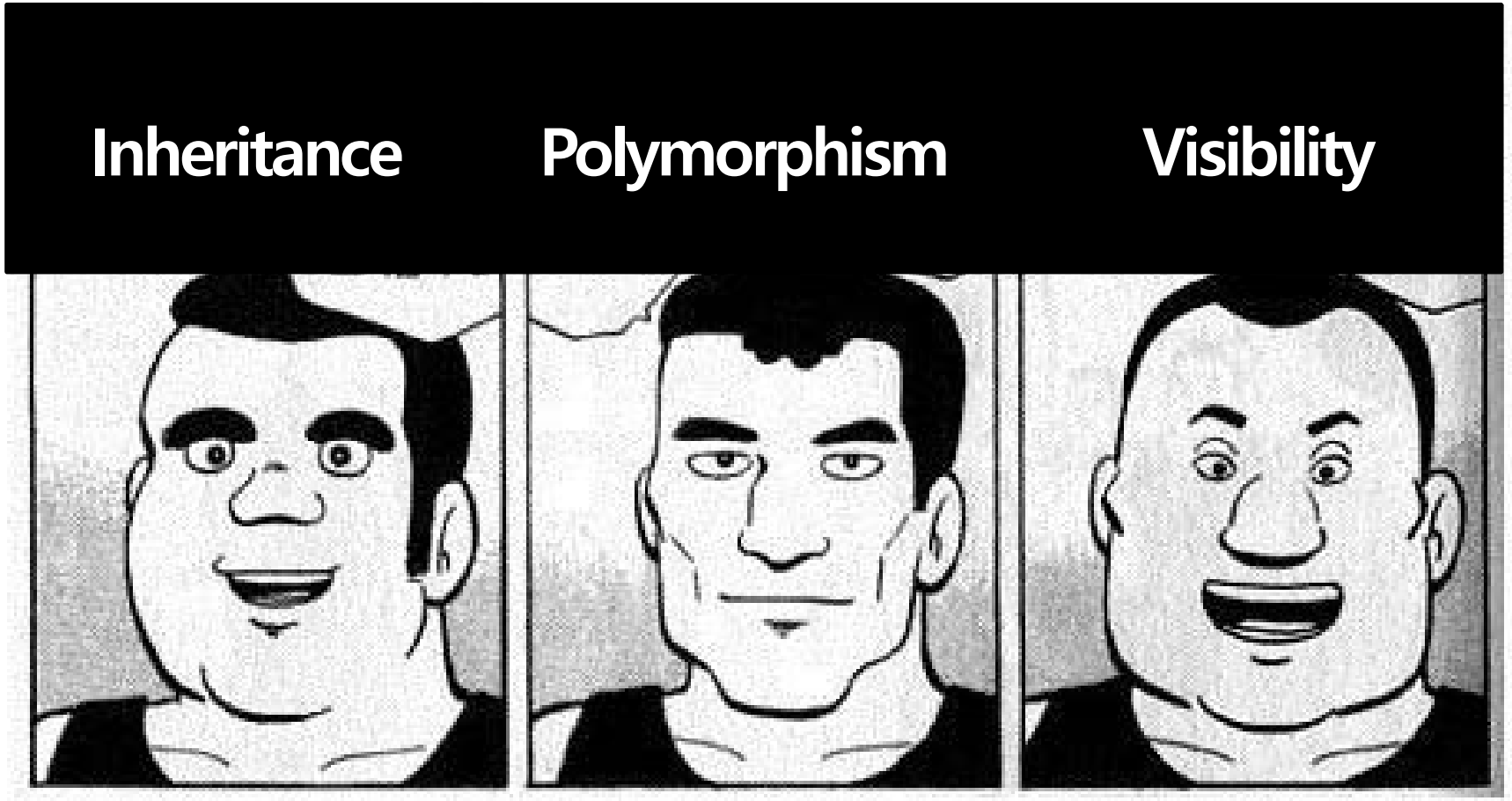
remove_note

get_number_of_pages

OOP characteristics

객체 지향 언어의 특징
실제 세상을 모델링

필요한 것 들



Inheritance

상속

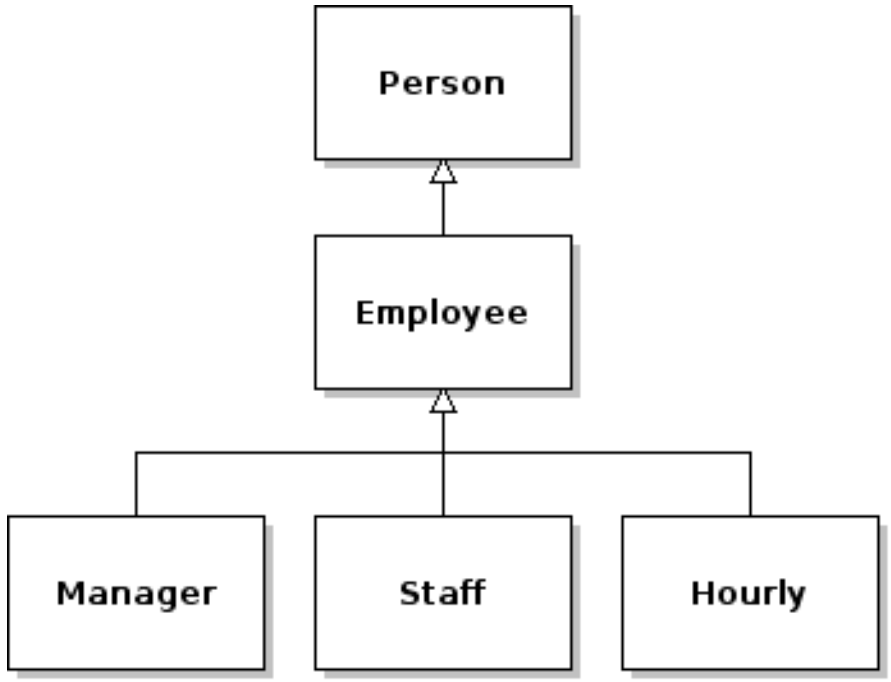
상속 (Inheritance)

- 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)
```



inheritance example


```
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.")
```

inheritance example


```
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.")
```

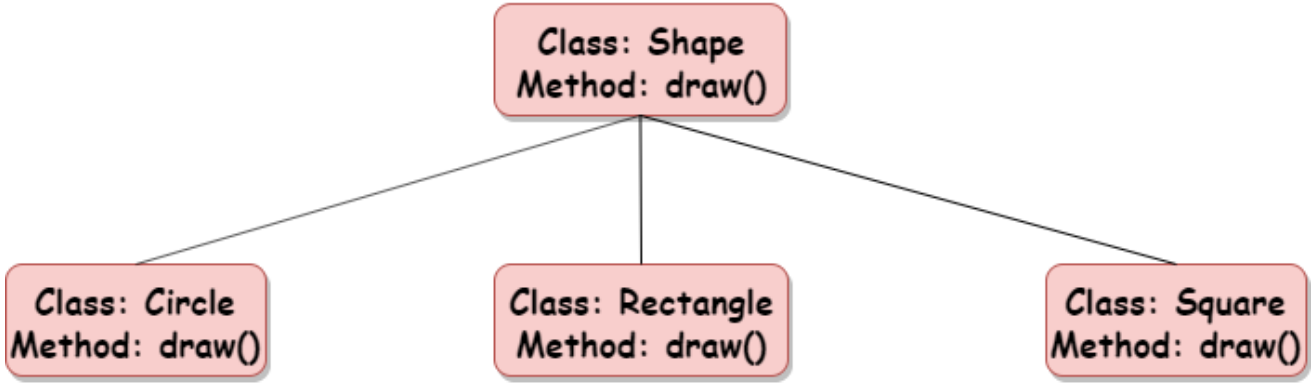
Polymorphism
다형성

Polymorphism overview

- 같은 이름 메소드의 내부 로직을 다르게 작성

- Dynamic Typing 특성으로 인해 파이썬에서는 같은 부모클래스의 상속에서 주로 발생함

- 중요한 OOP의 개념 그러나 너무 깊이 알 필요X



Polymorphism code


```
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())
```

×

Visibility
가시성

Visibility overview

- 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

[알아두면 상식] Encapsulation

Visibility Example 1

- 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)
```

Private 변수로 선언 타객체가 접근 못함

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 2

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

Visibility Example 2


```
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())
```

property decorator 숨겨진 변수를 반환하게 해줌

Property decorator로 함수를 변수처럼 호출

decorate

```
class Student:
    def __init__(self, name, marks):
        self.name = name
        self.marks = marks
        # self.gotmarks = self.name + ' obtained ' + self.marks + ' marks'

    @property
    def gotmarks(self):
        return self.name + ' obtained ' + self.marks + ' marks'
```

https://www.journaldev.com/14893/python-property-decorator

이해하기 위한 개념들

first-class objects
inner function
decorator

First-class objects

- 일등함수 또는 일급 객체
- 변수나 데이터 구조에 할당이 가능한 객체
- 파라메터로 전달이 가능 + 리턴 값으로 사용

파이썬의 함수는 일급함수

First-class objects


```
def square(x):
    return x * x

f = square
```

함수를 변수로 사용

```
f(5)
```

First-class objects


```
def square(x):
    return x * x

def cube(x):
    return x*x*x
```

함수를 파라메터로 사용

```
def formula(method, argument_list):
    return [method(value) for value in argument_list]
```

Inner function

- 함수 내에 또 다른 함수가 존재

```
def print_msg(msg):
    def printer():
        print(msg)
    printer()

print_msg("Hello, Python")
```

Inner function

- closures : inner function을 return값으로 반환

```
def print_msg(msg):
    def printer():
        print(msg)
    return printer

another = print_msg("Hello, Python")
another()
```

closure example


```
def tag_func(tag, text):
    text = text
    tag = tag

    def inner_func():
        return '<{0}>{1}<{0}>'.format(tag, text)

    return inner_func

h1_func = tag_func('title', "This is Python Class")
p_func = tag_func('p', "Data Academy")
```

decorator function

decorate

- 복잡한 클로져 함수를 간단하게!

```
def star(func):
    def inner(*args, **kwargs):
        print("*" * 30)
        func(*args, **kwargs)
        print("*" * 30)
    return inner

@star
def printer(msg):
    print(msg)
printer("Hello")
```

decorator function

decorate

- 복잡한 클로져 함수를 간단하게!

```
def star(func):
    def inner(*args, **kwargs):
        print("*" * 30)
        func(*args, **kwargs)
        print("*" * 30)
    return inner

def percent(func):
    def inner(*args, **kwargs):
        print("%" * 30)
        func(*args, **kwargs)
        print("%" * 30)
    return inner
```

×

```
@star
@percent
def printer(msg):
    print(msg)
printer("Hello")
```

```
******************************
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
Hello
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
******************************
```

decorator function

decorate

```
def generate_power(exponent):
    def wrapper(f):
        def inner(*args):
            result = f(*args)
            return exponent**result
        return inner
    return wrapper

@generate_power(2)
def raise_two(n):
    return n**2
```

```
print(raise_two(7))
```

```
562949953421312
```

End of Document
Thank You.