Fastcampus Web Programming SCHOOL

Class, OOP

Programming Paradigm

Imperative Programming

- Use statements to change a program's state.
- Assembly language, C, C++, Java, Go, Julia, Python, Ruby, ..

```c
#include<stdio.h>

int main(void){
    int a = 5;
    label:
        printf("Hello, world!\n");
        if(a==5){
            goto label;
        }
        printf("Bye, C.\n");
    return 0;
}
```

Procedural Programming

- Based on the concept of the unit and scope of an executable code statement.
- ALGOL, BASIC, C, C++, Java, Go, Julia, Python, Rust, javaScript, ..

```
from math import pi


def get_circle_area(radius):
    area = math.pi * radius ** 2
    return area

if __name__ == '__main__':
    user_radius = int(input("enter radius: "))
    get_circle_area(user_radius)
```

Object-Oriented Programming

- Based on the concept of objects, which contain data and code.
- Java, C++, C#, Python, R, PHP, javaScript, Objective-C, Swift, Scala, Kotlin, ..

```
from math import pi


class Circle:
    def __init__(self, radius):
        self.radius = radius

    def get_area(self):
        return pi * self.radius ** 2

tiny_circle = Circle(2)
tiny_circle.get_area()
```

Functional Programming

- constructed by applying and composing functions.
- Common Lisp, Scheme, Haskell, Clojure, Kotlin, Python, Go, Rust, Scala.

```
from math import pi


list(map(lambda a:pi*a**2, [i for i in range(1,10+1)]))
```

Scope

global, local, nonlocal

```
def outer_scope():
    def inner_local():
        msg = 'Fast'

    def inner_nonlocal():
        nonlocal msg
        msg = 'Campus'

    def inner_global():
        global msg
        msg = 'FastCampus'

    msg = 'Seong-su'
    inner_local()
    print('local', msg, id(msg))
    inner_nonlocal()
    print('nonlocal', msg, id(msg))
    inner_global()
    print('global', msg, id(msg))
```

Run

```
outer_scope()
print('global', msg, id(msg))
```

- local: outer_scope의 binding을 바꾸지 못함
- nonlocal: outer_scope의 binding을 바꿈
- global: 전역 binding 을 바꿈

Object-Oriented Programming

OOP

- 자료와 코드로 이루어진 객체 중심의 프로그래밍
- 코드의 중복을 줄이고, 직관적인 코드 이해를 목적으로 함.
- 강한 응집력(Strong Cohesion), 약한 결합력(Weak Coupling)을 지향
  - Cohesion: 기능 수행을 위한 요소와 개념의 뭉쳐있는 정도
  - Coupling: 어떤 요소의 다른 요소와의 연결, 의존관계

OOP의 특징

- Encapsulation
  - 구현한 것을 드러나지 않도록 함
  - Information hiding: public, _protected, __private
- Abstraction
  - 인터페이스로 클래스의 공통 특성을 묶어 표현
- Inheritance
  - 자식 클래스가 부모 클래스의 특성과 기능을 물려받음
- Polymorphism
  - 변수, method가 다른 상태를 가지는 것

SOLID

- Single Responsibility Principle
  - 한 클래스는 하나의 책임만 가져야 한다.(goto 19)
- Open/Closed Principle
  - 확장에는 열려있지만, 변경에는 닫혀 있어야 한다.(override)
- Liskov's Substitution Principle
  - 프로그램의 정확성을 깨트리지 않으면서 하위 타입의 인스턴스로 바꿀 수 있어야 한다.(is-A)
- Interface Segregation Principle
  - 사용하지 않는 method는 분리해야 한다.(Abstract)
- Dependency Inversion Principle
  - 추상화에 의존하고, 구체화에 의존하지 않는다.

구성요소

- Class: 같은 종류의 집단에 속하는 속성과 행의의 정의
- Object: Class의 Instance. 상위 Class의 속성과 함께 개별 특성과 Method를 가짐.
- Method: 클래스로 생성된 객체의 사용법. 객체의 속성을 조작.

Example

```
class Hero: # class
    health = 100 # class variable

    def __init__(self, name, weapon):
        self.name = name # instance variable
        self.weapon = weapon

    def attack(self): # instance method
        print("attack with {}".format(self.weapon))

IronMan = Hero('Iron Man', 'Impulse Gun') # instance
```

- `__init__` : initializer(instance 생성시 자동으로 호출되어 self로 instance object를 받음)

class

```
class ClassName:
    #statements
```

- CamelCase!!

class object

```
class SeeAttributes:
    """
    You can see docstring.
    """
    integer = 1024 # attribute

    def function(self): # attribute
        return 'fastcampus'
```

```
LetsSee = SeeAttributes()
LetsSee.integer
LetsSee.function
LetsSee.integer = 16
dir(LetsSee)
LetsSee.__doc__
```

Back to Hero..

```
Hulk = Hero('Hulk', 'Radioactive Fist')
Hulk.health = 10000
Hulk.health
```

class variable: Not for Mutable

```
class Hero: # class
    health = 100 # class variable
    inventory = []

    def __init__(self, name, weapon):
        self.name = name # instance variable
        self.weapon = weapon

    def attack(self): # instance method
        print("attack with {}".format(self.weapon))
    def save_item(self, item):
        self.inventory.append(item)
```

check inventory

```
IronMan = Hero('Iron Man', 'Impulse Gun')
Hulk = Hero('Hulk', 'Radioactive Fist')
IronMan.save_item('Fancy glasses')
Hulk.save_item('torn shirt')
IronMan.inventory
```

Right way to use Mutable object

```
class Hero: # class
    health = 100 # class variable

    def __init__(self, name, weapon, inventory):
        self.name = name # instance variable
        self.weapon = weapon
        self.inventory = []

    def attack(self): # instance method
        print("attack with {}".format(self.weapon))
    def save_item(self, item):
        self.inventory.append(item)
```

Deconstructor

```
def __del__(self):
    print("I love you 3000.")

del IronMan
```

Call Another Method using self

```
def save_item_multiple(self, item, num):
    for _ in range(num):
        self.save_item(item)
```

- self의 method attribute를 사용해 다른 method를 호출 가능!!

Hide Information

```
class SmartPhone:
    def __init__(self, ap, cam):
        # Public: Open to every class
        self.ap = ap
        self.cam = cam

    def _jail_break(self):
        # Protected: Open only to this or inherited from this
        return 'jail break complete'

    def __custom_firmware(self):
        # Private: Only this class
        return "You can't do this"
```

Practice(1)

다음 조건에 맞는 class Elevator를 설계하세요.

```
1. 목적(use_for): 화물용, 사람용
2. 고유번호(elevator_id): 임의 배정
3. 공간(space): []
4. 상승(go_up()): 출발층, 목적층
5. 하강(go_down()): 출발층, 목적층
6. 한계하중(max_cap)
7. 현재하중(cur_cap)
8. 현재상태(status): 점검중, 만원
```

Inheritance

```
class DerivedClassName(BaseClassName):
    #Statements
```

- 부모 클래스의 속성을 물려받아 새로운 인스턴스를 생성하는 것

Winner Winner, Chicken Dinner!

```
class Fried:
    def __init__(self, mixture, chicken):
        self.mixture = mixture
        self.chicken = chicken

    def place_into_fryer(self):
        print('chicken is now frying for 15 min..')


class Seasoned(Fried):
    def __init__(self, mixture, chicken, sauce='red chili'):
        Computer.__init__(self, mixture, chicken)
        self.sauce = sauce

    def place_into_fryer(self):
        print('chicken in now frying for 13 min..')

    def mix_with_sauce(self):
        print('mix with {}'.format(self.sauce))
```

Smart inheritance

```
class SmartPhone:
    def __init__(self, ap, cam):
        self.ap = ap
        self.cam = cam

    def open_ai(self):
        print('Hey, Kakao!')

    def __str__(self):
        return 'I am {}'.format(self.__class__.__name__)
```

iphone inheritance

```
class IPhone(SmartPhone):
    def __init__(self, ap, cam, touch_id):
        # with class name
        SmartPhone.__init__(self, ap, cam)
        self.touch_id = touch_id

    def open_ai(self): # override
        print('Hey, Siri!')

    def __str__(self):
        return super(IPhone, self).__str__()
```

galaxy inheritance

```
class Galaxy(SmartPhone):
    def __init__(self, ap, cam, sam_pay):
        # with super()
        super(Galaxy, self).__init__(ap, cam)
        self.sam_pay = sam_pay

    def open_ai(self):
        print('Hi, Bixby!')

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

Override

- 부모와 자식에 같은 이름의 함수 존재시, 자식의 함수가 우선
- Overload: 같은 공간(Namespace)에 같은 이름의 함수를 정의하는 것.(파이썬은 마지막으로 정의된 함수만 인정)

is-A (Inheritance)

- `IPhone` is a `SmartPhone`
- `Galaxy` is a `SmartPhone`
- Is `Iphone` `Galaxy`? (Nope)

Practice(2)

- 생활속에서 상속으로 표현 가능한 간단한 예를 클래스로 구현하세요.

has-A (Composition, Aggregation)

- `SmartPhone` has an `AP`
- `SmartPhone` has a `Cam`

Composition

```
class AP:
    pass


class Cam:
    pass


class SmartPhone:
    def __init__(self):
        self.ap = AP()
        self.cam = Cam()
```

Aggregation

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

    def fly(self):
        print('Flying..')


class Thor:
    def __init__(self):
        self.weapon = None

    def recall_hammer(self, weapon):
        self.weapon = weapon

    def throw_hammer(self):
        weapon = self.weapon
        self.weapon = None
        return weapon

    def fly(self):
        if self.weapon:
            self.weapon.fly()
        else:
            print("You can't do this")
```

Run Aggregation

```
Thor = Thor()
molnir = Hammer('molnir')
Thor.recall_hammer(molnir)
Thor.fly()
Thor.throw_hammer()
Thor.fly()
```

Final Practice

- 지금까지 배운 개념을 활용하여 블랙잭 게임의 딜러와 유저를 클래스로 구현하세요.