

Day39; 20221101

날짜	@2022년 11월 1일
유형	@2022년 11월 1일
태그	

GitHub - u8yes/Python

You can't perform that action at this time. You signed in with another tab or window. You signed out in another tab or window. Reload to refresh your session. Reload to refresh your session.

<https://github.com/u8yes/Python>

u8yes/Python



1 Contributor 0 Issues 1 Star 0 Forks

https://s3-us-west-2.amazonaws.com/secure.notion-static.com/52b535a5-ccf5-490d-85b4-1cb8dc071913/08._%EC%83%81%EC%86%8D.pdf


https://s3-us-west-2.amazonaws.com/secure.notion-static.com/80bf9d8e-879d-4767-b2ee-6076273b0071/chap07_Class_20221031.zip

Python은 참조 자료형을 선언할 때는 첫글자를 대문자로 작성해야한다.

Python은 ':' 뒤에 들여쓰기를 통해 지역변수라는 것을 보여준다.

Python은 (JAVA같은) 생성자를 __init__(self)로 선언한다.

this도 self로 정의한다.

 ':' 치면 나오는 아이콘 ㅋ

클래스 메서드에 사용되는 self

- 메소드가 소속되어 있는 객체 자체 의미.
- 클래스 외부에서는 변수명으로 객체를 다룰 수 있지만 내부에서는 객체를 지칭할 이름이 없으므로 self를 사용.
- java의 객체 자신을 가리키는 this와 같음.

Car.py

```
class Car: # DB 데이터를 관리하는 자바의 VO같은 역할을 하는 클래스
    def __init__(self): # 생성자를 정의함. # self는 시작 주소값을 가지고 있다.
        self.color = 0xFF0000 # 필드 선언 # self는 생략불가능(필드라는 표시의 의도) # 차량 색상(빨간색)
        # 생성자는 초기화를 해줌
        self.wheel_size = 16 # 바퀴의 크기
        self.displacement = 2000 # 엔진 배기량

    def forward(self):          # 전진
        pass                   # 기능은 보류(pass)

    def backward(self):         # 후진
        pass

    def turn_left(self):        # 좌회전
```

```

pass

def turn_right(self):      # 우회전
    pass

if __name__ == '__main__':
    my_car = Car()        # Car()는 'new' 메모리 할당하는 것과 같음 # Car()클래스의 객체 생성
    # 1)자료형 찾고 크기 계산해서 메모리 할당
    # 2)self 키워드에 주소값 저장
    # 3)생성자 호출
    # 4)(호출하려고)주소(self)값을 리턴해줌
    # my_car. 으로 접근할 수 있게 만들어줌.

    print('0x{:02X}'.format(my_car.color)) # 10진수로 나오기 때문에, :02X를 넣어줘서 16진수로 출력.
    print(my_car.wheel_size)
    print(my_car.displacement)

    my_car.forward() # 굳이 self 값을 안 넣음.
    my_car.backward()
    my_car.turn_left()
    my_car.turn_right()

```

0xFF0000

16

2000

static

heap의 특징은

new로 생성할 때 사용하고 나서 언제 소멸할지를 모름.

static은 명확하게 new 한 것이 메소드 수행 후 복귀하는 순간 바로 소멸시킴. (Stack에서 만들고 삭제되고, heap 영역으로 보내진 건 그대로 남아서 자바가 알아서 소멸시켜 줌. stack 영역이 지워졌다고 해서 바로 heap에서 지워지지 않는. 자바 안에 그런 알고리즘이 있어서 구현돼있다.)

그것이 클래스 이름으로 접근할 수 있게 한 이유. 클래스 메서드, 클래스 변수라고 하는 이유.

new를 생성할 때 바로 heap에서 생성.

ClassVar.py (**static** 변수)

```

class ClassVar: # static 과 같다. # 생성자가 쓰지 않아도 정상 구동됨. # 클래스 변수, 클래스 메서드
    # 전역변수로 사용 가능함.
    # static은 호출하면 할당했다가 끝나면 할당된 것을 삭제함.
    text_list = []          # 클래스 변수

    def add(self, text):
        self.text_list.append(text)

    def print_list(self):
        print(self.text_list)

if __name__ == '__main__':
    ClassVar.text_list.append('a')
    print(ClassVar.text_list)

    x = ClassVar()
    x.add('b')
    x.print_list()

    print(ClassVar.text_list)

    y = ClassVar()
    y.add('c')
    y.print_list()

    print(ClassVar.text_list)

```

['a']

['a', 'b']

['a', 'b']

['a', 'b', 'c']

['a', 'b', 'c']

InstanceVar.py (멤버변수)

```

class InstanceVar:
    def __init__(self):
        self.text_list = []          # 멤버변수(field)

    def add(self, text):              # 멤버 메서드
        self.text_list.append(text)

    def print_list(self):
        print(self.text_list)

if __name__ == '__main__':
    # InstanceVar.text_list.append('a') # error

    x = InstanceVar()
    x.add('a')
    x.print_list()          # ['a']
    print(x.text_list)     # ['a']

    y = InstanceVar()

```

```
y.add('b')
y.print_list()      # ['b']
```

['a']

['a']

['b']

ContactInfo.py

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

    def print_info(self):
        print(f'{self.name} : {self.email}')

if __name__ == '__main__':
    james = ContactInfo('야고보', 'u8yes@naver.com')
    simon = ContactInfo('베드로', 'u8yes@naver.com')

    james.print_info()
    simon.print_info()
```

야고보 : u8yes@naver.com

베드로 : u8yes@naver.com

Calculator.py

```
class Calculator:

    @staticmethod # java의 어노테이션 같이 정의 # @는 데코레이터라고 부름 # static 정의 데코
    def plus(x1, x2):      # self(자신의 주소값을 만들 때 생김)가 사라짐
        return x1 + x2

    @staticmethod
    def minus(x1, x2):
        return x1 - x2

    @staticmethod
    def multiply(x1, x2):
        return x1 * x2

    @staticmethod
    def divide(x1, x2):
        return x1 / x2

if __name__ == '__main__':
    print(f'{7} + {4} = {Calculator.plus(7,4)}')
    print(f'{7} - {4} = {Calculator.minus(7, 4)}')
```

```
print(f'{7} * {4} = {Calculator.multiply(7, 4)}')
```

```
print(f'{7} / {4} = {Calculator.divide(7, 4)}')
```

7 + 4 = 11

7 - 4 = 3

7 * 4 = 28

7 / 4 = 1.75

InstanceCounter.py

```
class InstanceCounter:
    count = 0

    def __init__(self):
        InstanceCounter.count += 1

    @classmethod
    def print_instance_count(cls): # self는 new로써 만들어지기 때문에 지움
        print(cls.count)

if __name__ == '__main__':
    x = InstanceCounter()
    x.print_instance_count()

    InstanceCounter.print_instance_count()

    y = InstanceCounter()
    y.print_instance_count()

    InstanceCounter.print_instance_count()

    InstanceCounter.count = 10
    InstanceCounter.print_instance_count()
```

1

1

2

2

10

protected는 다른 패키지로 있어도 부모의 상속이 있다면 접근할 수 있다.

public 같은 프로젝트 내에서는 어떤 위치에서라도 접근을 허용해줌.

HasPrivate.py

```

class HasPrivate:
    def __init__(self):
        self.public1 = "public1"
        self.__private1 = "private1" # '__' 변수 이름앞에 언더바2개를 붙이면 바로 private 특성을 가지게 된다.
        self.__private2_ = "private2"
        self.__public2__ = "public2" # 이름으로 public, private 참고

    def print_from_internal(self):
        print(self.public1)
        print(self.__private1)
        print(self.__private2_)
        print(self.__public2__)

if __name__ == '__main__':
    obj = HasPrivate()
    obj.print_from_internal()

    print("=====")
    print(obj.public1)
    # print(obj.__private1) # error - private 속성이기 때문에
    # print(obj.__private2_) # error - private 속성이기 때문에
    print(obj.__public2__)

```

public1

private1

private2

public2

=====

public1

public2

sqlD 1,2 과목 준비

https://www.youtube.com/watch?v=_dx3fPb766E&list=PLICujDgOz8x4JN2wHKbmIM8bFan-WaKj5

<https://youtu.be/iPxbcmhYQ2o>

baseinheritance.py

```

class Base: # 부모 클래스 # 상위 클래스
    def __init__(self):
        print(self) # 16진수 # 인스턴스 생성 요청할 때 시작 주소값을 알려줌
        # <__main__.Base object at 0x00000199CC224588>

        self.message = "Heaven World"

    def print_message(self):
        print(self.message)

class Derived(Base): # 상속의 관계 정의 해줌. # 자식 클래스 # 하위 클래스
    pass

if __name__ == '__main__':
    base = Base()
    base.print_message() # <__main__.Base object at 0x000001C086D24608>

    derived = Derived() # 부모를 상속받은 자식의 메서드 # 시작 주소값이 저장돼있기 때문에 부모로 바로 실행 가능
    derived.print_message() # <__main__.Derived object at 0x000001C086D24648>

```

polymorphism.py

```

# 다형성(Polymorphism)
# 상속의 조건하에서 자식의 인스턴스를 생성했을 때 1)자식의 자료형으로 바라볼 수 있고
# 2)부모의 자료형으로도 바라볼 수 있는 것.

class Suite:
    pass

class ArmorSuite:
    def armor(self):
        print("armored")

class IronMan(ArmorSuite):
    pass

def get_armored(suite):
    suite.armor()

if __name__ == '__main__':
    suite = ArmorSuite() # 참조 변수를 통해서 접근을 할 수 있다.
    get_armored(suite)

    iron_main = IronMan()
    get_armored(iron_main)

    suite = Suite()
    # get_armored(suite) # error # AttributeError: 'Suite' object has no attribute 'armor'

```


coalesce

미국식[ˌkoʊəˈles]  영국식[ˌkəʊəˈles] 

동사

(더 큰 덩어리로) 합치다 (=amalgamate)

The puddles had coalesced into a small stream. 

물웅덩이들이 합쳐져 작은 냇물을 이루고 있었다.

[영어사전 다른 뜻 2](#)

표 5-16 단일행 함수의 종류

종류	설명	주요 단일행 함수
NULL 관련 함수	NULL을 처리하기 위한 함수이다.	NVL, NULLIF, COALESCE

fieldinheritance.py

```
# 데이터 속성(field) 상속

class A:
    def __init__(self):
        print("A.__init__() 생성자 호출")
        self.message = "Heavenly"

class B(A):
    def __init__(self):
        print("B.__init__() 생성자 호출")

if __name__ == '__main__':
    obj = B()

    print(obj.message) # AttributeError: 'B' object has no attribute 'message'
```

