Python Basic

2021. 6

Soft Engineer Society

클래스(Class)

- 객체(Object)
  - 연관된 속성들과 동작들을 하나로 묶은 것.
  - 프로그래밍 대상을 마치 현실 세계의 사물처럼 표현하기 위한 것.

- 클래스(Class)
  - 객체를 생성하기 위해 변수와 메서드를 정의하는 일종의 틀.
  - 클래스를 통해 객체가 생성됨.

클래스의 생성

- 클래스의 생성

```
class 클래스명:
    내용

변수명 = 클래스명()
```

```
class Car:
    color = "blue"

new_car = Car()
print(new_car.color)
```

생성자

- 생성자
  - 객체의 초기값을 설정하기 위한 특수한 메서드.
  - 객체가 생성될 때 호출됨.

- 파이썬에서의 생성자
  - 특수 메서드 `__init__`을 사용함.
  - 일반적으로 생성자에서 멤버를 생성함.
  - 생성자의 첫 번째 인자는 무조건 자기 자신을 의미하는 `'self'`.
  - 클래스 내에서 멤버를 호출할 때 `self.`멤버명의 형태로 사용.
  - 두 개 이상의 생성자를 만들 수 없음.

► 생성자

```
class 클래스명:
    def __init__(self, 멤버의 초기값):
        멤버 생성 및 초기화 코드
```

```
class Car:
    def __init__(self, color):
        self.color = color

    def print_color(self):
        print(self.color)

new_car = Car("Blue")
print(new_car.color)

new_car.print_color()
Car.print_color(new_car)
```

생성자

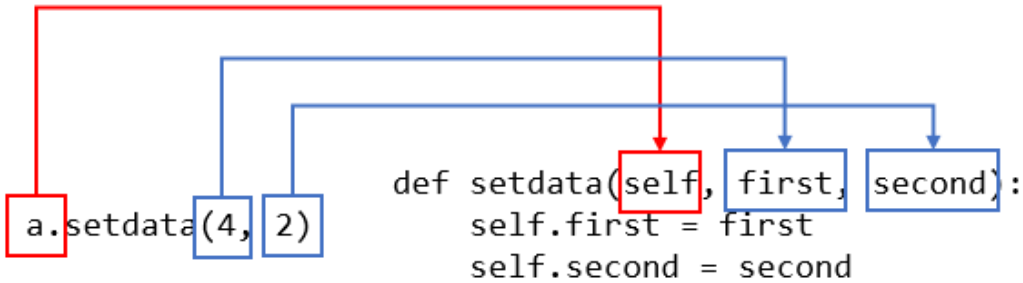
```
class FourCal :
    def setdata(self, first, second) :
        self.first  = first
        self.second = second

a = FourCal()                       # 객체 생성
a.setdata(3, 4)                     # 첫 번째 호출 방법

b = FourCal()
FourCal.setdata(b, 5, 6)            # 두 번째 호출 방법, b 생략 불가

print(a.first, a.second)
print(b.first, b.second)
```

※ self변수는 객체명을 받는 용도이며, 생략 불가



상속

- 문법

```
class 자식클래스명(부모클래스명) :
```

- [예]

```
class Car() :
    def exclaim(self) :
        print("I'm a car")

class Yogo(Car) :
    pass

car = Car()
print(car.exclaim())        # I'm a car 출력

yogo = Yogo()
print(yogo.exclaim())       # I'm a car 출력, 메서드 상속됨
```

상속

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

class MDPerson(Person) :
    def __init__(self, name) :
        self.name = "Doctor " + name

class JDPerson(Person) :
    def __init__(self, name) :
        self.name = name +", Esquire"

person = Person('Fudd')
doctor = MDPerson('Fudd')
lawer  = JDPerson('Fudd')

print(person.name)      # Fudd 출력
print(doctor.name)      # Doctor Fudd 출력
print(lawer.name)       # Fudd, Esquire 출력
```

- super()
  - 부모클래스의 생성자 호출

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

class EmailPerson(Person) :
    def __init__(self, name, email) : # 메서드 오버라이딩
        super().__init__(name)  # 부모클래스의 __init__() 호출
        self.email = email

bob = EmailPerson('박길동', 'bob@frapples.com')
print(bob.name)
print(bob.email)
```

※ 파이썬은 다중 상속을 지원한다!

getter, setter

- 접근지정자
  - 파이썬은 멤버의 접근지정자가 없음(기본적으로 public).
  - 변수명 앞에 '__'를 붙여 private, '_'를 붙여 protected, 아무것도 없을 때 public을 나타내기로 암묵적으로 약속함.

- getter, setter
  - 멤버명과 다른 이름의 getter와 setter를 설정하여 멤버를 private처럼 표현할 수 있음.
  - property() 함수를 이용하는 방법과 데커레이터(@)를 이용하는 방법이 있음.

getter, setter

- 일반적인 getter, setter

```
class Date:
    def __init__(self, month):
        self.inner_month = month
    def getMonth(self):
        return self.inner_month
    def setMonth(self, month):
        self.inner_month = month

today = Date(7)
today.setMonth(8)
print(today.getMonth())
```

getter, setter

- property() 함수를 이용한 getter, setter

```
class Date:
    def __init__(self, month):
        self.inner_month = month
    def getMonth(self):
        return self.inner_month
    def setMonth(self, month):
        self.inner_month = month
    month = property(getMonth, setMonth)

today = Date(7)
today.month = 8
print(today.month)
```

getter, setter

- 데커레이터(@)를 이용한 getter, setter

```
class Date:
    def __init__(self, month):
        self.inner_month = month
    @property
    def month(self):
        return self.inner_month
    @month.setter
    def month(self, month):
        self.inner_month = month

today = Date(7)
today.month = 8
print(today.month)
```

getter, setter

► 멤버명에 '__'를 붙일 경우

```
class Date:
    def __init__(self, month):
        self.__month = month
    @property
    def month(self):
        return self.__month
    @month.setter
    def month(self, month):
        self.__month = month

today = Date(7)
today.__month = 8
today._Date__month = 9
print(today.month)
```

클래스 메서드

- 클래스 메서드
  - 클래스 전체가 공유하는 메서드.
  - @classmethod 데커레이터를 사용해 표현함.
  - 클래스를 인수(cls)로 받음.

```
class Car:
    count = 0
    def __init__(self, name):
        self.name = name
        Car.count += 1
    @classmethod
    def outcount(cls):
        print(cls.count)

rangerover = Car("레인지로버")
renegade = Car("레니게이드")
Car.outcount()
print(rangerover.count)
print(renegade.count)
```

정적 메서드

- 정적 메서드
  - 클래스에 존재하지만 클래스(또는 객체)와 직접적인 연관이 없는 유틸리티 메서드.
  - @staticmethod 데커레이터를 사용해 표현함.
  - self나 cls를 인수로 받지 않음.

```
class Car:
    @staticmethod
    def hello():
        print("안녕하세요?")
    count = 0
    def __init__(self, name):
        self.name = name
        Car.count += 1
    @classmethod
    def outcount(cls):
        print(cls.count)


Car.hello()
```

클래스 메서드와 정적 메서드의 차이

- 클래스 메서드 vs 정적 메서드
  - 클래스 메서드는 인수로 전달받은 클래스가 중심.
  - 정적 메서드는 해당 메서드를 선언한 클래스가 중심.

```
class Parent:
    f_name = "Kim"

    @classmethod
    def class_name(cls):
        return cls.f_name
    @staticmethod
    def static_name():
        return Parent.f_name

class Child(Parent):
    f_name = "Lee"

son = Child()
print(son.class_name())
print(son.static_name())
```

연산자 메서드

- 연산자 메서드
  - 클래스에서 연산자 메서드를 정의하여 해당 객체에 대해 연산자를 통한 연산 가능.
  - 연산자의 동작을 임의로 정의하는 기능을 연산자 오버로딩이라 함.

연산자	메서드	연산자	메서드
==	__eq__	!=	__ne__
<	__lt__	>	__gt__
<=	__le__	>=	__ge__
+	__add__, __radd__	-	__sub__, __rsub__
*	__mul__, __rmul__	/	__div__, __rdiv__
%	__mod__, __rmod__	/	__truediv__, __rtruediv__
**	__pow__, __rpow__	//	__floordiv__, __rfloordiv__
<<	__lshift__, __rlshift__	>>	__rshift__, __rrshift__

► 연산자 메서드

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

    def __eq__(self, other):
        return self.name == other.name

hong = Human("홍길동")
gil = Human("홍길동")
jeon = Human("전우치")
print(hong == gil)
print(hong == jeon)
```

특수 메서드

- 특수 메서드
  - 특정 함수에 의해 객체가 사용될 때 호출되는 메서드.

메서드	관련 함수	설명
__str__	str(), print()	객체에 대한 설명을 반환한다.
__len__	len()	객체의 길이를 반환한다.

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

    def __str__(self):
        return "이름: %s" % self.name

hong = Human("홍길동")
print(hong)
```

추상 클래스

- 추상 클래스
  - 구현되지 않은 추상 메서드를 갖는 클래스.
  - 자식 클래스에서 해당 추상 메서드를 반드시 구현하도록 강제함.
  - 구현하지 않으면 객체를 생성할 때 에러 발생.
  - abc 모듈과 ABCMeta 객체의 상속이 필요함.
  - @abstractmethod 데커레이터를 사용해서 추상 메서드 정의.

```
from abc import ABCMeta, abstractmethod
class 추상클래스명(metaclass=ABCMeta):
    @abstractmethod
    def 추상메소드명(self):
        pass
```

► 추상 클래스

```
from abc import ABCMeta, abstractmethod
class Hello(metaclass=ABCMeta):
    @abstractmethod
    def say(self):
        pass

class Hi(Hello):
    def say(self):
        return "Hi"

h = Hi()
h.say()
```