

# 파이썬의 클래스(class)

이번 장에서는 파이썬에서 클래스, 객체, 상속 등을 알아봅니다.



1. 클래스와 객체
2. 클래스 정의하기
3. 객체 만들기
4. 메소드 호출하기
5. 프라이빗 멤버와 퍼블릭 멤버
6. 클래스 멤버
7. 특별한 메소드 : 연산자 메소드, 클래스 메소드, 정적 메소드
8. 상속
9. 기타





## □ 객체(object)

- 객체는 데이터와 이를 처리하기 위한 메소드의 묶음

## □ 클래스(class)

- 클래스는 특정한 종류의 객체를 만드는 형틀(template)
- 클래스로부터 만들어지는 객체를 그 클래스의 인스턴스(instance)라고도 함

### <참고>

- 파이썬에서는 모든 것이 객체로 구현됨

(예) 정수도 객체이며, 정수 객체는 `__add__()` 메소드를 가짐.

```
>> (1).__add__(2)
```

```
3
```



## 2. 클래스 정의하기

### □ 클래스 정의하기

- 클래스 정의는 `class` 키워드로 시작하며, 클래스 이름은 보통 대문자로 시작함
- 데이터 초기화 및 메소드들로 구성됨
- 데이터 초기화를 위해 `__init__()` 이라는 특별한 메소드를 가짐
  - ✓ 객체가 생성될 때 자동 호출됨
- 메소드는 함수처럼 `def` 키워드를 사용해서 정의함

```
class 클래스이름 :
```

```
    def __init__(self, 초기값) :           ➡ 매개변수 self를 가짐  
        멤버 초기화
```

```
    메소드 정의                               ➡ 함수처럼 def 키워드를 사용해서 정의함
```



## 2. 클래스 정의하기

### □ 클래스 예

```
class Human :  
    def __init__(self, age, name) :  
        self.age = age  
        self.name = name  
  
    def intro(self) :  
        print(str(self.age) + “살” + self.name + “입니다” )
```

```
kim = Human(29, “김유신” )
```

👉 Human의 객체 kim을 만듦

```
kim.intro()
```

```
hong = Human(25, “홍길동” )
```

👉 Human의 객체 hong을 만듦

```
hong.intro()
```



### 3. 객체 만들기

#### □ 객체 생성

- 클래스 이름으로 호출하면, 객체가 생성됨
- 이때, `__init__` 메소드가 호출되면서 객체를 초기화함

```
kim = Human(29, "김유신" )  
      ↙      ↘      ↘  
__init__(self, age, name) :
```

- ① kim을 self로 전달
- ② 29를 age로 전달
- ③ "김유신"을 name으로 전달



## 4. 메소드 호출하기

### □ 객체에서 메소드 호출하기

- 객체를 만든 후, "객체 이름.메소드" 형식으로 호출

```
class Player :  
    def __init__(self, name, height) :  
        self.number = 100  
        self.name = name  
        self.height = height
```

```
    def print_info(self) :  
        print(self.number)  
        print(self.name)  
        print(self.height)
```

← 메소드는 매개변수 self를 가짐

```
a = Player('gildong', '178')  
a.print_info()
```

👉 Player의 객체 a를 만들



□ 다음과 같이 은행 계좌를 나타내는 BankAccount 클래스를 구현하시오.

- `__init__` 메소드는 다음 매개변수를 초기화함
  - `balance` : 잔액 (int)
  - `name` : 소유자 이름 (string)
  - `number` : 계좌 번호 (int)
- `withdraw()` : 출금 메소드  
돈 액수를 입력 매개변수로 받아 통장 잔액에서 빼고, 잔액값을 반환
- `deposit()` : 입금 메소드  
돈 액수를 입력 매개변수로 받아 통장 잔액에 더하고, 잔액값을 반환





- 앞에서 만든 BankAccount 클래스를 사용하여 프로그램을 만드시오.

소유자 이름이 "홍길동", 계좌번호는 "123456", 잔액은 "10000"인 계좌 객체 a1 을 만들  
계좌 a1에 30000원을 입금





## 5. 프라이빗 멤버와 퍼블릭 멤버

### □ 프라이빗 멤버 (private member)

- 클래스 안에서만 접근 가능한 멤버로서 작성법으로 구분함
- 작성 규칙
  - 두 개의 밑줄로 시작해야 함
  - 마지막에 밑줄이 한 개까지 허용됨 (2개 이상이면 퍼블릭 멤버가 됨)

### □ 퍼블릭 멤버 (public member)

- 클래스 안과 밖에서 모두 접근 가능한 멤버
- 프라이빗 멤버 이외의 모든 멤버가 해당됨



## 5. 프라이빗 멤버와 퍼블릭 멤버

```
class TestPrivate :  
    def __init__(self) :  
        self.a = 100          ← a는 public 멤버  
        self.__b = 200       ← b는 private 멤버  
  
    def print_member(self) :  
        print(self.a)  
        print(self.__b)
```

```
obj = TestPrivate()  
obj.print_member()  
100  
200  
print(obj.a)  
  
print(obj.__b)      ➡ 에러: 멤버 __b는 객체의 메소드 호출을 통해서만 접근가능
```



### □ 클래스 멤버

- 클래스의 메소드 밖에 만드는 변수
- 객체 생성과 무관하게 사용할 때 (클래스 이름으로도 직접 접근 가능)

```
class MyClass :  
    var = "안녕하세요!!"           # 클래스 멤버  
  
    def sayHello(self) :  
        msg1 = "안녕"  
        self.msg2 = "Hi"  
        print(msg1)  
        print(self.var)           # 객체의 var를 새로 정의하여 사용  
  
obj = MyClass()  
print(obj.var)                    # 객체를 통한 클래스 멤버 접근  
obj.sayHello()                   # 객체의 메소드를 통한 접근  
print(MyClass.var)               # 클래스 이름으로 접근
```



- 다음 프로그램을 실행하고 출력 결과를 확인하고 설명하시오.

```
class CalCounter :  
    count = 0                                # 클래스 멤버  
    def __init__(self) :  
        CalCounter.count += 1              # 클래스 멤버인 count의 값을 접근  
  
    def print_count(self) :  
        print(self.count)  
  
a = CalCounter()  
a.print_count()  
b = CalCounter()  
b.print_count()
```



- 다음 프로그램을 실행해보고, 앞의 프로그램과 비교하여 설명하시오.

```
class CalCounter :  
    count = 0                                # 클래스 멤버  
    def __init__(self) :  
        self.count += 1                    # 객체의 count를 접근  
  
    def print_count(self) :  
        print(self.count)  
  
a = CalCounter()  
a.print_count()  
b = CalCounter()  
b.print_count()
```



## 5. 특별한 메소드 : 연산자 메소드

연산자	메서드	우변일 때의 메서드
==	<code>__eq__</code>	
!=	<code>__ne__</code>	
<	<code>__lt__</code>	
>	<code>__gt__</code>	
<=	<code>__le__</code>	
>=	<code>__ge__</code>	
+	<code>__add__</code>	<code>__radd__</code>
-	<code>__sub__</code>	<code>__rsub__</code>
*	<code>__mul__</code>	<code>__rmul__</code>
/	<code>__div__</code>	<code>__rdiv__</code>
/(division 임포트)	<code>__truediv__</code>	<code>__rtruediv__</code>
//	<code>__floordiv__</code>	<code>__rfloordiv__</code>
%	<code>__mod__</code>	<code>__rmod__</code>
**	<code>__pow__</code>	<code>__rpow__</code>
<<	<code>__lshift__</code>	<code>__rshift__</code>
>>	<code>__rshift__</code>	<code>__lshift__</code>

연산자를 사용하여  
객체끼리 연산이  
가능하도록  
클래스에서 구현하는  
메소드



## 7. 특별한 메소드 : 연산자 메소드

### □ \_\_eq\_\_ 메소드의 구현 예

```
class Human :  
    def __init__(self, age, name) :  
        self.age = age  
        self.name = name  
  
    def __eq__(self, other) :  
        return self.age == other.age and self.name == other.name  
  
kim = Human(29, "홍길동")  
sang = Human(29, "홍길동")  
moon = Human(30, "이순신")  
print(kim==sang)  
print(kim==moon)
```





1. 앞에서 만든 Human 클래스에서 `__equ__` 메소드는 이름과 나이가 모두 같은 지를 반환해주고 있다. 나이는 제외하고 이름만 같은 지를 반환하도록 수정하시오.
2. Human 클래스에서 `__le__` 메소드를 다음과 같이 추가하시오.  
`self`의 나이가 `other`의 나이보다 작거나 같은 지를 반환함.
3. 앞의 프로그램에서, `1 + kim` 을 하면 `kim`의 나이에 1을 추가하도록 구현하시오.  
이를 위해, `__radd__` 메소드를 추가해야 함





## 7. 특별한 메소드 : 클래스 메소드

### □ 클래스 메소드 (class method)

- @classmethod 로 시작하며, 매개변수로 cls를 사용해야 함
- 클래스 이름을 사용하여 직접 호출 (메소드 내에서 클래스 멤버 접근 가능)

```
class CalCounter :  
    object_count = 0  
    def __init__(self) :  
        CalCounter.object_count += 1
```

☞ 생성한 객체 개수

**@classmethod**

```
def print_objectcount(cls) :  
    print(cls.object_count)
```

☞ cls : 클래스 자신을 매개변수로 받음

```
a = CalCounter()  
CalCounter.print_objectcount()  
b = CalCounter()  
CalCounter.print_objectcount()
```

☞ 1이 출력됨

☞ 2가 출력됨



## 7. 특별한 메소드 : 정적 메소드

### □ 정적 메소드 (static method)

- @staticmethod 로 시작하며, 매개변수에 self 없이 정의함
- 클래스 이름을 사용하여 직접 호출 또는 객체를 생성하여 객체로 호출함

```
class Calculator :
```

```
    @staticmethod
```

```
    def plus(a, b) :  
        return a+b
```

매개변수 self를 사용하지 않음

```
    @staticmethod
```

```
    def minus(a, b) :  
        return a-b
```

```
result = Calculator.plus(3, 5)
```

☞ 클래스 이름으로 호출

```
print(result)
```

```
cal1 = Calculator()
```

```
print(cal1.plus(3, 5))
```

☞ 객체의 메소드로 호출



## 8. 상속 (inheritance)

### □ 다른 클래스의 속성과 메소드를 물려받는 것

- 기반(base) 클래스 : 물려주는 클래스 (부모 클래스)
- 파생(derived) 클래스 : 물려받는 클래스 (자식 클래스)

```
class 기반 클래스:
```

```
    # 멤버 정의
```

```
class 파생 클래스(기반 클래스) :
```

```
    # 기반 클래스의 모든 것을 물려받게 됨
```

```
    # 단, 프라이빗 멤버는 물려받지 않음.
```



## 8. 상속

```
class BaseA :  
    def print_string(self) :  
        print( 'This is base' )
```

```
class DerivedA(BaseA) :  
    pass
```

☞ BaseA 클래스의 메소드를 물려받음

```
a = BaseA()  
a.print_string()
```

```
b = DerivedA()  
b.print_string()
```

☞ 상속받은 print\_string 메소드를 호출





## 8. 상속

- 다음 프로그램을 실행하고 출력 결과를 확인하시오

```
class Add :  
    def add(self, a, b) :  
        return a+b
```

```
class Calculator(Add) :  
    def sub(self, a, b) :  
        return a-b
```

☞ Add 클래스의 메소드를 물려받음

```
c = Calculator()  
print(c.add(1,2))  
print(c.sub(2,1))
```



### □ super()

- 부모 클래스의 객체 역할을 하는 프록시를 반환하는 내장함수
- 부모 클래스의 메소드를 호출할 때 사용할 수 있음

```
class BaseA :  
    def __init__(self) :  
        print( 'BaseA.__init__()' )
```

```
class B(BaseA) :  
    def __init__(self) :  
        print( 'B.__init__()' )  
        super().__init__()
```

☞ BaseA 클래스의 메소드를 물려받음

☞ BaseA의 \_\_init\_\_( )을 호출

```
b = B()
```

객체 b를 만들면서 클래스 B의 \_\_init\_\_( )을 호출하게 됨.  
또한, 부모 클래스의 \_\_init\_\_( )을 호출하고 있음



### □ 다중 상속

- 여러 부모 클래스로부터 상속을 받는 것

```
class A :  
    pass
```

```
class B :  
    pass
```

```
class C :  
    pass
```

```
class D(A,B,C) :  
    pass
```

☞ 클래스 A, B, C로부터 다중 상속





- 다음 프로그램을 실행하고 출력 결과를 확인하시오

```
class Add :  
    def add(self, a, b) :  
        return a+b
```

```
class Multiply :  
    def multiply(self, a, b) :  
        return a*b
```

```
class Calculator(Add, Multiply) : ➡ 클래스 Add, Multiply로부터 다중 상속  
    def sub(self, a, b) :  
        return a-b
```

```
c = Calculator()  
print(c.add(1,2))  
print(c.multiply(2,10))
```



### □ 오버라이딩 (overriding)

- 부모 클래스로부터 상속받은 메소드를 다시 정의하는 것

```
class A :  
    def print_str(self) :  
        print( "A" )
```

```
class B(A) :  
    def print_str(self) :  
        print( "B" )
```

- ☞ 클래스 A를 상속
- ☞ 클래스 A를 상속했으나,  
print\_str 메소드를 오버라이딩함

```
A().print_str()           // A의 객체를 만들면서 print_str 호출  
B().print_str()           // B의 객체를 만들면서 print_str 호출
```



## 8. 상속

- 다음 프로그램을 실행하고 출력 결과를 확인하고, 설명하시오.

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

    def intro(self) :
        print(str(self.age) + "살" + self.name + "입니다")

class Student(Human) :
    def school(self, s) :
        self.school = s

    def intro(self) :
        print(self.name + "의 학교는 " + self.school + "입니다")

kim = Human(29, "홍길동")
kim.intro()

kim2 = Student(25, "이순신")
kim2.school("명지대")
kim2.intro()
```



### □ `__str__()` 메소드

- 객체를 print할 때 출력할 값을 반환해 줌

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

```
    def __str__(self) :  
        return "이름 %s, 나이 %d"%(self.name, self.age)
```

```
kim = Human(29, "홍길동")
```

```
print(kim)           # 객체 kim을 print -> __str__ 메소드가 호출됨
```



### □ `__call__()` 메소드

- 객체를 함수 호출하듯이 사용할 수 있게 해주며, 이때 `__call__()` 메소드의 내용이 실행됨

```
class A :  
    def __init__(self) :  
        print( 'A' )  
  
    def __call__(self) :  
        print( '__call__()' )
```

```
b = A()          # 객체를 생성 → __init__ 호출 → A를 출력  
b()              # __call__() 호출
```