



## 9. 객체 지향 프로그래밍과 클래스

파이썬 프로그래밍

# Contents

## ❖ 객체 지향 프로그래밍

- 객체와 클래스
- 파이썬에서 코딩하며 객체를 지향한다.

## ❖ 클래스의 정의

- `__init__()` 메소드를 이용한 초기화
- `self`에 대하여
- 정적 메소드와 클래스 메소드
- 클래스 내부에게만 열려있는 프라이빗 멤버

## ❖ 상속

- `super()`
- 다중상속
- 오버라이딩

## ❖ 데코레이터 : 함수를 꾸미는 객체

## ❖ `for`문으로 순회를 할 수 있는 객체 만들기

- 이터레이터와 순회 가능한 객체
- 제네레이터

## ❖ 상속의 조건 : 추상 기반 클래스



# 시작하기 전에

## ❖ 객체 지향 프로그래밍 (Object Oriented Programming)

- 객체를 우선으로 생각해서 프로그래밍하는 것
- 클래스 기반의 객체 지향 프로그래밍 언어는 클래스를 기반으로 객체 만들고, 그러한 객체를 우선으로 생각하여 프로그래밍함
  - 클래스 (class)
  - 객체 (object)



❖ 객체(Object) = 속성(Attribute) + 기능(Method)

❖ 속성은 사물의 특징

- 예) 자동차의 속성 : 바디의 색, 바퀴의 크기, 엔진의 배기량

❖ 기능은 어떤 것의 특징적인 동작

- 예) 자동차의 기능 : 전진, 후진, 좌회전, 우회전

❖ 속성과 기능을 들어 자동차를 묘사하면?

- “18인치의 바퀴를 가진 2,000cc의 빨간 차는 전진, 후진, 좌회전, 우회전의 기능이 있다.”



# 객체

## ❖ 데이터 (data)

- 예시 - 딕셔너리로 객체 만들기

```
01  # 학생 리스트를 선언합니다.
02  students = [
03      { "name": "윤인성", "korean": 87, "math": 98, "english": 88, "science": 95 },
04      { "name": "연하진", "korean": 92, "math": 98, "english": 96, "science": 98 },
05      { "name": "구지연", "korean": 76, "math": 96, "english": 94, "science": 90 },
06      { "name": "나선주", "korean": 98, "math": 92, "english": 96, "science": 92 },
07      { "name": "윤아린", "korean": 95, "math": 98, "english": 98, "science": 98 },
08      { "name": "윤명월", "korean": 64, "math": 88, "english": 92, "science": 92 }
09  ]
10
11  # 학생을 한 명씩 반복합니다.
12  print("이름", "총점", "평균", sep="\t")
```



# 객체

```
13 for student in students:
14     # 점수의 총합과 평균을 구합니다.
15     score_sum = student["korean"] + student["math"] + \
16         student["english"] + student["science"]
17     score_average = score_sum / 4
18     # 출력합니다.
19     print(student["name"], score_sum, score_average, sep="\t")
```

| 실행결과 |     |       |
|------|-----|-------|
| 이름   | 총점  | 평균    |
| 윤인성  | 368 | 92.0  |
| 연하진  | 384 | 96.0  |
| 구지연  | 356 | 89.0  |
| 나선주  | 378 | 94.5  |
| 윤아린  | 389 | 97.25 |
| 윤명월  | 336 | 84.0  |



# 객체

## ❖ 객체 (object)

- 여러 가지 속성 가질 수 있는 모든 대상
- 예시 - 객체를 만드는 함수

```
01  # 딕셔너리를 리턴하는 함수를 선언합니다.  
02  def create_student(name, korean, math, english, science):  
03      return {  
04          "name": name,  
05          "korean": korean,  
06          "math": math,  
07          "english": english,  
08          "science": science  
09      }  
10
```



# 객체

```
11 # 학생 리스트를 선언합니다.
12 students = [
13     create_student("윤인성", 87, 98, 88, 95),
14     create_student("연하진", 92, 98, 96, 98),
15     create_student("구지연", 76, 96, 94, 90),
16     create_student("나선주", 98, 92, 96, 92),
17     create_student("윤아린", 95, 98, 98, 98),
18     create_student("윤명월", 64, 88, 92, 92)
19 ]
20
21 # 학생을 한 명씩 반복합니다.
22 print("이름", "총점", "평균", sep="\t")
23 for student in students:
24     # 점수의 총합과 평균을 구합니다.
25     score_sum = student["korean"] + student["math"] + \
26         student["english"] + student["science"]
27     score_average = score_sum / 4
28     # 출력합니다.
29     print(student["name"], score_sum, score_average, sep="\t")
```





# 객체

- 학생을 매개변수로 받는 형태의 함수로 만들면 코드가 더 균형 잡히게 됨

```
01 # 딕셔너리를 리턴하는 함수를 선언합니다.
02 def create_student(name, korean, math, english, science):
03     return {
04         "name": name,
05         "korean": korean,
06         "math": math,
07         "english": english,
08         "science": science
09     }
10
11 # 학생을 처리하는 함수를 선언합니다.
12 def student_get_sum(student):
13     return student["korean"] + student["math"] + \
14         student["english"] + student["science"]
15
16 def student_get_average(student):
17     return student_get_sum(student) / 4
18
19 def student_to_string(student):
20     return "{}\t{}\t{}".format(
21         student["name"],
22         student_get_sum(student),
23         student_get_average(student))
```

→ 01~23행까지  
학생 객체와  
관련된 부분



# 객체

```
24
25 # 학생 리스트를 선언합니다.
26 students = [
27     create_student("윤인성", 87, 98, 88, 95),
28     create_student("연하진", 92, 98, 96, 98),
29     create_student("구지연", 76, 96, 94, 90),
30     create_student("나선주", 98, 92, 96, 92),
35 # 학생을 한 명씩 반복합니다.
36 print("이름", "총점", "평균", sep="\t")
37 for student in students:
38     # 출력합니다.
39     print(student_to_string(student))
```

→ 25~39행까지  
객체를 활용하는



# 클래스 선언하기

## ❖ 클래스 (class)

- 객체를 조금 더 효율적으로 생성하기 위해 만들어진 구문

```
class 클래스 이름:  
    클래스 내용
```

```
인스턴스 이름(변수 이름) = 클래스 이름() → 생성자 함수라고 부릅니다.
```

- 인스턴스 (instance)
  - 생성자 사용하여 이러한 클래스 기반으로 만들어진 객체



# 클래스 선언하기

# 클래스를 선언합니다.

```
class Student:  
    pass
```

# 학생을 선언합니다.

```
student = Student()
```

# 학생 리스트를 선언합니다.

```
students = [  
    Student(),  
    Student(),  
    Student(),  
    Student(),  
    Student(),  
    Student()  
]
```



## ❖ 생성자 (constructor)

- 클래스 이름과 같은 함수

```
class 클래스 이름:  
    def __init__(self, 추가적인 매개변수):  
        pass
```

- 클래스 내부의 함수는 첫 번째 매개변수로 반드시 self 입력해야 함
  - self : '자기 자신' 나타내는 디렉터리
  - self.<식별자> 형태로 접근



# 생성자

# 클래스를 선언합니다.

```
class Student:
```

```
    def __init__(self, name, korean, math, english, science):
```

```
        self.name = name
```

```
        self.korean = korean
```

```
        self.math = math
```

```
        self.english = english
```

```
        self.science = science
```

# 학생 리스트를 선언합니다.

```
students = [
```

```
    Student("윤인성", 87, 98, 88, 95),
```

```
    Student("연하진", 92, 98, 96, 98),
```

```
    Student("구지연", 76, 96, 94, 90),
```

```
    Student("나선주", 98, 92, 96, 92),
```

```
    Student("윤아린", 95, 98, 98, 98),
```

```
    Student("윤명월", 64, 88, 92, 92)
```

```
]
```

# Student 인스턴스의 속성에 접근하는 방법

```
students[0].name
```

```
students[0].korean
```

```
students[0].math
```

```
students[0].english
```

```
students[0].science
```



# 메소드

## ❖ 메소드 (method)

- 클래스가 가지고 있는 함수

```
class 클래스 이름:  
    def 메소드 이름(self, 추가적인 매개변수):  
        pass
```



- 예시 - 클래스 내부에 함수 선언하기

```
01  # 클래스를 선언합니다.
02  class Student:
03      def __init__(self, name, korean, math, english, science):
04          self.name = name
05          self.korean = korean
06          self.math = math
07          self.english = english
08          self.science = science
09
10      def get_sum(self):
11          return self.korean + self.math + \
12              self.english + self.science
13
14      def get_average(self):
15          return self.get_sum() / 4
16
```





# 메소드

```
17     def to_string(self):
18         return "{}\t{}\t{}".format(\
19             self.name,\
20             self.get_sum(),\
21             self.get_average())
22
23 # 학생 리스트를 선언합니다.
24 students = [
25     Student("윤인성", 87, 98, 88, 95),
26     Student("연하진", 92, 98, 96, 98),
27     Student("구지연", 76, 96, 94, 90),
28     Student("나선주", 98, 92, 96, 92),
29     Student("윤아린", 95, 98, 98, 98),
30     Student("윤명월", 64, 88, 92, 92)
31 ]
32
33 # 학생을 한 명씩 반복합니다.
34 print("이름", "총점", "평균", sep="\t")
35 for student in students:
36     # 출력합니다.
37     print(student.to_string())
```

| 실행결과 |     |       |
|------|-----|-------|
| 이름   | 총점  | 평균    |
| 윤인성  | 368 | 92.0  |
| 연하진  | 384 | 96.0  |
| 구지연  | 356 | 89.0  |
| 나선주  | 378 | 94.5  |
| 윤아린  | 389 | 97.25 |
| 윤명월  | 336 | 84.0  |



# 키워드로 정리하는 핵심 포인트

- ❖ **객체** : 속성을 가질 수 있는 모든 것 의미
- ❖ **객체 지향 프로그래밍 언어** : 객체를 기반으로 프로그램 만드는 프로그래밍 언어
- ❖ **추상화** : 복잡한 자료, 모듈, 시스템 등으로부터 핵심적인 개념 또는 기능을 간추려 내는 것
- ❖ **클래스** : 객체를 쉽고 편리하게 생성하기 위해 만들어진 구문
- ❖ **인스턴스** : 클래스를 기반으로 생성한 객체
- ❖ **생성자** : 클래스 이름과 같은 인스턴스 생성할 때 만드는 함수
- ❖ **메소드** : 클래스가 가진 함수



### ❖ 다음과 같이 묘사한 자동차를 코드로 표현하면... (1)

- “18인치의 바퀴를 가진 2,000cc의 빨간 차는 전진, 후진, 좌회전, 우회전의 기능이 있다.”

```
color = 0xFF0000    # 바디의 색
wheel_size = 18      # 바퀴의 크기
displacement = 2000  # 엔진 배기량
```

```
def forward(): # 전진
    pass
```

```
def backward(): # 후진
    pass
```

```
def turn_left(): # 좌회전
    pass
```

```
def turn_right(): # 우회전
    pass
```

**아직 속성과 기능이  
혼여져있음.**



### ❖ 다음과 같이 묘사한 자동차를 코드로 표현하면... (2)

- “18인치의 바퀴를 가진 2,000cc의 빨간 차는 전진, 후진, 좌회전, 우회전의 기능이 있다.”

```
class Car:
```

Car 클래스의 정의 시작을 알립니다.

```
    def __init__(self):
```

```
        self.color = 0xFF0000
```

# 바디의 색

```
        self.wheel_size = 18
```

# 바퀴의 크기

```
        self.displacement = 2000
```

# 엔진 배기량

Car 클래스 안에 차의 색, 바퀴 크기, 배기량을 나타내는 변수를 정의합니다.

```
    def forward(self): # 전진
```

```
        pass
```

```
    def backward(self): # 후진
```

```
        pass
```

```
    def turn_left(self): # 좌회전
```

```
        pass
```

```
    def turn_right(self): # 우회전
```

```
        pass
```

Car 클래스 안에 전진, 후진, 좌회전, 우회전 함수를 정의합니다.

- ❖ 앞에서 만든 Car 클래스는 자료형
- ❖ Car 클래스의 객체는 다음과 같이 정의함

```
num = 123      # 자료형:int, 변수: num  
my_car = Car() # 자료형: Car 클래스, 객체: my_car
```

- ❖ 객체 대신 인스턴스(Instance)라는 용어를 사용하기도 함.
  - 클래스가 설계도, 객체는 그 설계를 바탕으로 실체화한 것이라는 뜻에서 유래한 용어
  - 객체뿐 아니라 변수도 인스턴스라고 부름. 자료형을 메모리에 실체화한 것이 변수이기 때문임.



## 객체 지향 프로그래밍 – 파이썬에서 코딩하며 객체를 지향한다.

### ❖ 컴퓨터의 업그레이드가 용이한 이유?

- 컴퓨터 부품간의 결합도(coupling)가 낮기 때문.

### ❖ 이와 비해 태블릿과 스마트폰의 업그레이드는 거의 불가능.

- 부품간의 결합도가 매우 높기 때문.

### ❖ 결합도는 한 시스템 내의 구성 요소간의 의존성을 나타내는 용어.

- 소프트웨어에서도 결합도가 존재함.
- 예) A() 함수를 수정했을 때 B() 함수의 동작에 부작용이 생긴다면 이 두 함수는 **강한 결합도**를 보인다고 할 수 있음.
- 예) A() 함수를 수정했는데도 B() 함수가 어떤 영향도 받지 않는다면 이 두 함수는 **약한 결합**으로 이루어져 있다고 할 수 있음.

### ❖ 클래스 안에 같은 목적과 기능을 위해 묶인 코드 요소(변수, 함수)는 객체 내부에서만 강한 응집력을 발휘하고 객체 외부에 주는 영향은 줄이게 됨.



# 클래스의 정의

## ❖ 클래스는 다음과 같이 `class` 키워드를 이용하여 정의.

```
class 클래스이름:  
    코드블록
```

## ❖ 클래스의 코드블록은 변수와 메소드(Method)로 이루어짐.

- 기능(Method) : 객체 지향 프로그래밍에서 사물의 동작을 나타냄.
- 메소드(Method) : 객체 지향 프로그래밍의 기능에 대응하는 파이썬 용어. 함수와 거의 동일한 의미이지만 메소드는 클래스의 멤버라는 점이 다름.
- 함수(Function) : 일련의 코드를 하나의 이름 아래 묶은 코드 요소.

## ❖ 객체의 멤버(메소드와 데이터 속성에 접근하기)

- 객체의 멤버에 접근할 때는 점(.)을 이용.

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

my\_car의 멤버에 접근하게 해줍니다.

## ❖ 예제 : 09/Car.py

```
class Car:
    def __init__(self):
        self.color = 0xFF0000    # 바디의 색
        self.wheel_size = 16     # 바퀴의 크기
        self.displacement = 2000 # 엔진 배기량
```

```
    def forward(self): # 전진
        pass
```

```
    def backward(self): # 후진
        pass
```

```
    def turn_left(self): # 좌회전
        pass
```

```
    def turn_right(self): # 우회전
        pass
```

#Car 클래스 정의 종료. 아래는 Car 클래스의 인스턴스를 정의하고 사용하는 코드

```
if __name__ == '__main__':
    my_car = Car()

    print('0x{:02X}'.format(my_car.color))
    print(my_car.wheel_size)
    print(my_car.displacement)

    my_car.forward()
    my_car.backward()
    my_car.turn_left()
    my_car.turn_right()
```

### • 실행 결과

```
>Car.py
0xFF0000
16
2000
```





# 클래스의 정의 - `__init__()` 메소드를 이용한 초기화

## ❖ `__init__()`

- 객체가 생성된 후 가장먼저 호출되는 메소드
- “초기화하다”는 뜻의 initialize를 줄여서 붙여진 이름.

## ❖ 예제 : 09/InstanceVar.py

- 실행 결과

```
class InstanceVar:
    def __init__(self):
        self.text_list = []

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

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

if __name__ == '__main__':
    a = InstanceVar()
    a.add('a')
    a.print_list() # ['a'] 출력을 기대

    b = InstanceVar()
    b.add('b')
    b.print_list() # ['b'] 출력을 기대
```

```
>InstanceVar.py
['a']
['b']
```



# 클래스의 정의 - \_\_init\_\_() 메소드를 이용한 초기화

## ❖ 매개변수를 입력받는 \_\_init\_\_() 메소드의 예

```
class ContactInfo:  
    def __init__(self, name, email):  
        self.name = name  
        self.email = email  
  
sanghyun = ContactInfo('박상현', 'seanlab@gmail.com')
```

name      email

## ❖ 예제 : 09/ContactInfo.py

### • 실행 결과

```
class ContactInfo:  
    def __init__(self, name, email):  
        self.name = name  
        self.email = email  
  
    def print_info(self):  
        print('{0} : {1}'.format(self.name, self.email))  
  
if __name__ == '__main__':  
    sanghyun = ContactInfo('박상현', 'seanlab@gmail.com')  
    hanbit = ContactInfo('hanbit', 'noreply@hanb.co.kr')  
  
    sanghyun.print_info()  
    hanbit.print_info()
```

```
>ContactInfo.py  
박상현 : seanlab@gmail.com  
hanbit : noreply@hanb.co.kr
```



## 클래스의 정의 - self에 대하여

- ❖ 파이썬의 메소드에 사용되는 self가 가리키는 “자신” 은 바로 메소드가 소속되어 있는 객체

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

sanghyun = ContactInfo('박상현', 'seanlab@gmail.com')
```

- ❖ ContactInfo 외부에서는 sanghyun이라는 이름으로 객체를 다룰 수 있음.
- ❖ 내부에서는 sanghyun처럼 객체를 지칭할 수 있는 이름이 없기 때문에 self가 도입되었음.



# 어떤 클래스의 인스턴스인지 확인하기

## ❖ `isinstance()` 함수

- 객체가 어떤 클래스로부터 만들어졌는지 확인

```
isinstance(인스턴스, 클래스)
```

```
# 클래스를 선언합니다.
```

```
class Student:
```

```
    def __init__(self):
```

```
        pass
```

```
# 학생을 선언합니다.
```

```
student = Student()
```

```
# 인스턴스 확인하기
```

```
print("isinstance(student, Student):", isinstance(student, Student))
```

```
isinstance(students[0], Student): True
```



# 어떤 클래스의 인스턴스인지 확인하기

## ❖ isinstance() 함수의 다양한 활용

- 예시 - 리스트 내부에 여러 종류의 인스턴스 들어있을 때, 인스턴스들을 구분하며 속성과 기능 사용

```
01 # 학생 클래스를 선언합니다.
02 class Student:
03     def study(self):
04         print("공부를 합니다.")
05
06 # 선생님 클래스를 선언합니다.
07 class Teacher:
08     def teach(self):
09         print("학생을 가르칩니다.")
10
11 # 교실 내부의 객체 리스트를 생성합니다.
12 classroom = [Student(), Student(), Teacher(), Student(), Student()]
13
14 # 반복을 적용해서 적절한 함수를 호출하게 합니다.
15 for person in classroom:
16     if isinstance(person, Student):
17         person.study()
18     elif isinstance(person, Teacher):
19         person.teach()
```

실행결과

```
공부를 합니다.
공부를 합니다.
학생을 가르칩니다.
공부를 합니다.
공부를 합니다.
```



# 특수한 이름의 메소드

## ❖ 다양한 보조 기능들

- \_\_<이름>\_\_() 형태
- 특수한 상황에 자동으로 호출되도록 만들어짐



# 특수한 이름의 메소드

- 예시 - `__str__()` 함수

```
01  # 클래스를 선언합니다.
02  class Student:
03      def __init__(self, name, korean, math, english, science):
04          self.name = name
05          self.korean = korean
06          self.math = math
07          self.english = english
08          self.science = science
09
10      def get_sum(self):
11          return self.korean + self.math + \
12                 self.english + self.science
13
14      def get_average(self):
15          return self.get_sum() / 4
16
17      def __str__(self):
18          return "{}\t{}\t{}".format(
19              self.name,
20              self.get_sum(),
21              self.get_average())
```

→ `__str__()`이라는 이름으로  
함수를 선언했습니다.



# 특수한 이름의 메소드

```
28     Student("나선주", 98, 92, 96, 92),
29     Student("윤아린", 95, 98, 98, 98),
30     Student("윤명월", 64, 88, 92, 92)
31 ]
32
33 # 출력합니다.
34 print("이름", "총점", "평균", sep="\t")
35 for student in students:
36     print(str(student))
```

→ str() 함수의 매개변수로 넣으면  
student의 \_\_str\_\_ 함수가 호출됩니다.

| 실행결과 |     |       |
|------|-----|-------|
| 이름   | 총점  | 평균    |
| 윤인성  | 368 | 92.0  |
| 연하진  | 384 | 96.0  |
| 구지연  | 356 | 89.0  |
| 나선주  | 378 | 94.5  |
| 윤아린  | 389 | 97.25 |
| 윤명월  | 336 | 84.0  |

이와 같이 \_\_str\_\_() 함수 정의하면 str() 함수 호출할 때  
\_\_str\_\_() 함수가 자동으로 호출





# 특수한 이름의 메소드

| 이름 | 영어                    | 설명     |
|----|-----------------------|--------|
| eq | equal                 | 같다     |
| ne | not equal             | 다르다    |
| gt | greater than          | 크다     |
| ge | greater than or equal | 크거나 같다 |
| lt | less than             | 작다     |
| le | less than or equal    | 작거나 같다 |

- 예시 - 크기 비교 함수

```
01  # 클래스를 선언합니다.
02  class Student:
03      def __init__(self, name, korean, math, english, science):
04          self.name = name
05          self.korean = korean
06          self.math = math
07          self.english = english
08          self.science = science
09
```



# 특수한 이름의 메소드

```
10     def get_sum(self):
11         return self.korean + self.math + \
12             self.english + self.science
13
14     def get_average(self):
15         return self.get_sum() / 4
16
17     def __str__(self, student):
18         return "{}\t{}\t{}".format(
19             self.name,
20             self.get_sum(student),
21             self.get_average(student))
22
23     def __eq__(self, value):
24         return self.get_sum() == value.get_sum()
25     def __ne__(self, value):
26         return self.get_sum() != value.get_sum()
27     def __gt__(self, value):
28         return self.get_sum() > value.get_sum()
29     def __ge__(self, value):
30         return self.get_sum() >= value.get_sum()
```

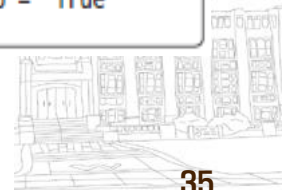


# 특수한 이름의 메소드

```
31     def __lt__(self, value):
32         return self.get_sum() < value.get_sum()
33     def __le__(self, value):
34         return self.get_sum() <= value.get_sum()
35
36     # 학생 리스트를 선언합니다.
37     students = [
38         Student("윤인성", 87, 98, 88, 95),
39         Student("연하진", 92, 98, 96, 98),
40         Student("구지연", 76, 96, 94, 90),
41         Student("나선주", 98, 92, 96, 92),
42         Student("윤아린", 95, 98, 98, 98),
43         Student("윤명월", 64, 88, 92, 92)
44     ]
45
46     # 학생을 선언합니다.
47     student_a = Student("윤인성", 87, 98, 88, 95),
48     student_b = Student("연하진", 92, 98, 96, 98),
49
50     # 출력합니다.
51     print("student_a == student_b = ", student_a == student_b)
52     print("student_a != student_b = ", student_a != student_b)
53     print("student_a > student_b = ", student_a > student_b)
54     print("student_a >= student_b = ", student_a >= student_b)
55     print("student_a < student_b = ", student_a < student_b)
56     print("student_a <= student_b = ", student_a <= student_b)
```

실행결과

```
student_a == student_b = False
student_a != student_b = True
student_a > student_b = False
student_a >= student_b = False
student_a < student_b = True
student_a <= student_b = True
```



# 클래스의 정의 - 정적 메소드와 클래스 메소드

## ❖ 인스턴스 메소드 - 인스턴스(객체)에 속한 메소드

- 인스턴스 메소드가 “인스턴스에 속한다”라는 표현은 “인스턴스를 통해 호출가능하다.”라는 뜻

## ❖ 정적 메소드와 클래스 메소드는 클래스에 귀속

## ❖ 정적 메소드

- @staticmethod 데코레이터로 수식
- self 키워드 없이 정의

```
class 클래스이름:
```

```
    @staticmethod
```

```
    def 메소드이름( 매개변수 ):
```

```
        pass
```

@staticmethod 데코레이터로 수식합니다.

self 매개변수는 사용하지 않습니다.

## ❖ 다른 언어와 다르게 정적 메소드 임에도 인스턴스에서 접근이 가능



# 클래스의 정의 - 정적 메소드와 클래스 메소드

## ❖ 예제 : 09/Calculator.py(정적 메소드)

```
class Calculator:

    @staticmethod
    def plus(a, b):
        return a+b

    @staticmethod
    def minus(a, b):
        return a-b

    @staticmethod
    def multiply(a, b):
        return a*b

    @staticmethod
    def divide(a, b):
        return a/b

if __name__ == '__main__':
    print("{0} + {1} = {2}".format(7, 4, Calculator.plus(7, 4)))
    print("{0} - {1} = {2}".format(7, 4, Calculator.minus(7, 4)))
    print("{0} * {1} = {2}".format(7, 4, Calculator.multiply(7, 4)))
    print("{0} / {1} = {2}".format(7, 4, Calculator.divide(7, 4)))
```

### • 실행 결과

```
>Calculator.py
7 + 4 = 11
7 + 4 = 3
7 + 4 = 28
7 + 4 = 1.75
```

# 클래스의 정의 - 정적 메소드와 클래스 메소드

## ❖ 클래스 메소드

- @classmethod 데코레이터로 수식
- cls 매개변수 사용

```
class 클래스이름:  
    # ...
```

```
    @classmethod  
    def 메소드이름(cls):  
        pass
```

클래스 메소드를 정의하기 위해서는...  
1. @classmethod 데코레이터를 앞에 붙여줍니다.

2. 메소드의 매개변수를 하나 이상 정의합니다.

## ❖ 실습 1

```
>>> class TestClass:  
        @classmethod  
        def print_TestClass(cls):  
            print(cls)
```

```
>>> TestClass.print_TestClass()  
<class '__main__.TestClass'>  
>>> obj = TestClass()  
>>> obj.print_TestClass()  
<class '__main__.TestClass'>
```

클래스를 통한 클래스 메소드 호출

인스턴스를 통한 클래스 메소드 호출

# 클래스의 정의 - 정적 메소드와 클래스 메소드

## ❖ 예제 : 09/InstanceCounter.py(클래스 메소드)

```
class InstanceCounter:  
    count = 0  
    def __init__(self):  
        InstanceCounter.count += 1
```

@classmethod

```
def print_instance_count(cls):  
    print(cls.count)
```

print\_instance\_count() 메소드는 InstanceCounter의 클래스 변수인 count를 출력합니다.

```
if __name__ == '__main__':  
    a = InstanceCounter()  
    InstanceCounter.print_instance_count()  
  
    b = InstanceCounter()  
    InstanceCounter.print_instance_count()  
  
    c = InstanceCounter()  
    c.print_instance_count()
```

### • 실행 결과

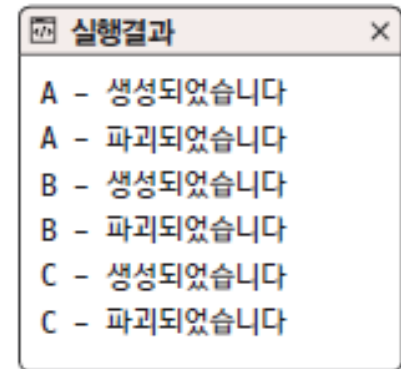
```
>InstanceCounter.py  
1  
2  
3
```

# 가비지 컬렉터

## ❖ 가비지 컬렉터 (garbage collector)

- 더 사용할 가능성이 없는 데이터를 메모리에서 제거하는 역할
- 예시 - 변수에 저장하지 않은 경우

```
01 class Test:
02     def __init__(self, name):
03         self.name = name
04         print("{} - 생성되었습니다".format(self.name))
05     def __del__(self):
06         print("{} - 파괴되었습니다".format(self.name))
07
08 Test("A")
09 Test("B")
10 Test("C")
```





# 가비지 컬렉터

- 예시 - 변수에 데이터 저장한 경우

```
01 class Test:
02     def __init__(self, name):
03         self.name = name
04         print("{} - 생성되었습니다".format(self.name))
05     def __del__(self):
06         print("{} - 파괴되었습니다".format(self.name))
07
08 a = Test("A")
09 b = Test("B")
10 c = Test("C")
```

**실행결과**

A - 생성되었습니다  
B - 생성되었습니다  
C - 생성되었습니다  
A - 파괴되었습니다  
B - 파괴되었습니다  
C - 파괴되었습니다



## 클래스의 정의 - 클래스 내부에게만 열려있는 프라이빗 멤버

- ❖ 클래스도 코드 블록을 가지므로 “안” 과 “밖” 개념이 존재함
- ❖ 다음 코드에서 MyClass 클래스의 “안” 은 상자로 표시한 부분이고 “밖” 은 상자로 표시된 부분을 제외한 나머지 코드를 말함.

```
class YourClass:  
    pass  
  
class MyClass:  
    def __init__(self):  
        self.message = "Hello"  
  
    def some_method(self):  
        print(self.message)  
  
obj = MyClass()  
obj.some_method()
```

- ❖ 프라이빗(Private) 멤버 : 클래스 내부에서만 접근이 가능한 멤버
- ❖ 퍼블리(Public) 멤버 : 안과 밖 모두에서 접근이 가능한 멤버



# 클래스의 정의 - 클래스 내부에게만 열려있는 프라이빗 멤버

## ❖ 프라이빗 멤버 명명 규칙

- 두 개의 밑줄 `__` 이 접두사여야 한다. 예) `__number`
- 접미사는 밑줄이 한 개까지만 허용된다. 예) `__number_`

```
>>> class HasPrivate:
    def __init__(self):
        self.public = "Public."
        self.__private = "Private."

    def print_from_internal(self):
        print(self.public)
        print(self.__private)
```

```
>>> obj = HasPrivate()
>>> obj.print_from_internal()
Public.
Private.
```

```
>>> print(obj.public)
Public.
>>> print(obj.__private)
```

```
Traceback (most recent call last):
  File "<pyshell#46>", line 1, in <module>
    print(obj.__private)
AttributeError: 'HasPrivate' object has no attribute '__private'
```

HasPrivate의 `print_from_internal()` 함수는 `public`, `__private` 두 데이터 속성에 자유롭게 접근할 수 있습니다. 같은 클래스의 멤버끼리니까요.

HasPrivate 객체 외부에서는 `__private` 데이터 속성에 접근할 수 없습니다. `__private` 데이터 속성이 아예 존재하지 않는 것처럼 보이기 때문입니다.

# 프라이빗 변수와 게터/세터

## ❖ 프라이빗 변수

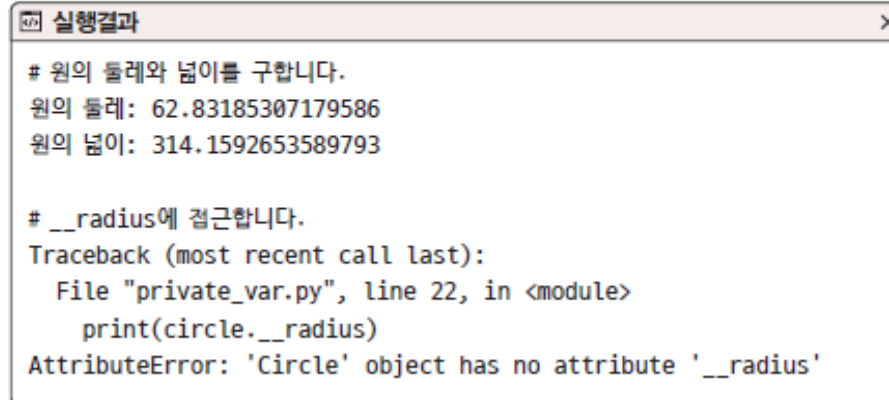
- 변수를 마음대로 사용하는 것 방지
- \_\_<변수 이름> 형태로 인스턴스 변수 이름 선언

```
01  # 모듈을 가져옵니다.
02  import math
03
04  # 클래스를 선언합니다.
05  class Circle:
06      def __init__(self, radius):
07          self.__radius = radius
08      def get_circumference(self):
09          return 2 * math.pi * self.__radius
10      def get_area(self):
11          return math.pi * (self.__radius ** 2)
12
```



# 프라이빗 변수와 게터/세터

```
13  # 원의 둘레와 넓이를 구합니다.
14  circle = Circle(10)
15  print("# 원의 둘레와 넓이를 구합니다.")
16  print("원의 둘레:", circle.get_circumference())
17  print("원의 넓이:", circle.get_area())
18  print()
19
20  # __radius에 접근합니다.
21  print("# __radius에 접근합니다.")
22  print(circle.__radius)
```



```
실행결과
# 원의 둘레와 넓이를 구합니다.
원의 둘레: 62.83185307179586
원의 넓이: 314.1592653589793

# __radius에 접근합니다.
Traceback (most recent call last):
  File "private_var.py", line 22, in <module>
    print(circle.__radius)
AttributeError: 'Circle' object has no attribute '__radius'
```



# 프라이빗 변수와 게터/세터

## ❖ 게터 (getter) 와 세터 (setter)

- 프라이빗 변수 값 추출하거나 변경할 목적으로 간접적으로 속성에 접근하도록 하는 함수
- 예시

```
01  # 모듈을 가져옵니다.  
02  import math  
03  
04  # 클래스를 선언합니다.  
05  class Circle:  
06      def __init__(self, radius):  
07          self.__radius = radius  
08      def get_circumference(self):  
09          return 2 * math.pi * self.__radius  
10      def get_area(self):  
11          return math.pi * (self.__radius ** 2)  
12
```



# 프라이빗 변수와 게터/세터

```
13     # 게터와 세터를 선언합니다.
14     def get_radius(self):
15         return self.__radius
16     def set_radius(self, value):
17         self.__radius = value
18
19     # 원의 둘레와 넓이를 구합니다.
20     circle = Circle(10)
21     print("# 원의 둘레와 넓이를 구합니다.")
22     print("원의 둘레:", circle.get_circumference())
23     print("원의 넓이:", circle.get_area())
24     print()
25
26     # 간접적으로 __radius에 접근합니다.
27     print("# __radius에 접근합니다.")
28     print(circle.get_radius())
29     print()
30
31     # 원의 둘레와 넓이를 구합니다.
32     circle.set_radius(2)
33     print("# 반지름을 변경하고 원의 둘레와 넓이를 구합니다.")
34     print("원의 둘레:", circle.get_circumference())
35     print("원의 넓이:", circle.get_area())
```

**실행결과**

```
# 원의 둘레와 넓이를 구합니다.
원의 둘레: 62.83185307179586
원의 넓이: 314.1592653589793

# __radius에 접근합니다.
10

# 반지름을 변경하고 원의 둘레와 넓이를 구합니다.
원의 둘레: 12.566370614359172
원의 넓이: 12.566370614359172
```



# 프라이빗 변수와 게터/세터

- 이와 같이 함수 사용해 값 변경하면 여러 가지 처리 추가할 수 있음
  - ex) set\_radius() 함수에 다음과 같은 코드 추가하여 \_\_radius에 할당할 값을 양의 숫자로만 한정

```
def set_radius(self, value):  
    if value <= 0:  
        raise TypeError("길이는 양의 숫자여야 합니다.")  
    self.__radius = value
```





# 프라이빗 변수와 게터/세터

## ❖ 데코레이터를 사용한 게터와 세터

- 파이썬 프로그래밍 언어에서 제공하는 게터와 세터 만들고 사용하는 기능
  - 변수 이름과 같은 함수 정의하고 위에 @property와 @<변수 이름>.setter 데코레이터 붙이기

```
01  # 모듈을 가져옵니다.
02  import math
03
04  # 클래스를 선언합니다.
05  class Circle:
    # ...생략...
13  # 게터와 세터를 선언합니다.
14  @property
15  def radius(self):
16      return self.__radius
17  @radius.setter
18  def radius(self, value):
19      if value <= 0:
20          raise TypeError("길이는 양의 숫자여야 합니다.")
```



# 프라이빗 변수와 게터/세터

```
21         self.__radius = value
22
23     # 원의 둘레와 넓이를 구합니다.
24     print("# 데코레이터를 사용한 Getter와 Setter")
25     circle = Circle(10)
26     print("원래 원의 반지름: ", circle.radius)
27     circle.radius = 2
28     print("변경된 원의 반지름: ", circle.radius)
29     print()
30
31     # 강제로 예외를 발생시킵니다.
32     print("# 강제로 예외를 발생시킵니다.")
33     circle.radius = -10
```

실행결과

```
# 데코레이터를 사용한 Getter와 Setter
원래 원의 반지름:  10
변경된 원의 반지름:  2

# 강제로 예외를 발생시킵니다.
Traceback (most recent call last):
  File "deco01.py", line 33, in <module>
    circle.radius = -10
  File "deco01.py", line 20, in radius
    raise TypeError("길이는 양의 숫자여야 합니다.")
TypeError: 길이는 양의 숫자여야 합니다.
```



❖ 다음과 같은 클래스를 설계하고, 객체를 생성하여 기능을 수행하세요.

에어컨

멤버변수

- 온도(정수), 전원상태(bool)

멤버함수

- 전원을 ON/OFF
- 온도를 입력
- 온도를 반환
- 현재온도를 출력
- 온도를 증가
- 온도를 감소

# 참고사항

클래스명

Air\_Conditioner

멤버변수명(프라이빗 변수로)

temperature

power

멤버함수명

togglePower()

temperate() - getter

temperate() - setter

print\_temperature()

increase()

decrease()



# 연습문제

❖ 다음과 같은 클래스를 설계하고, 객체를 생성하여 기능을 수행하세요.

```
Class Air_Conditioner:
    .. (생략) ..

if __name__ == "__main__":
    # 에어컨 인스턴스 생성
    my_air = Air_Conditioner()
    # 에어컨의 전원을 켭니다.
    my_air.togglePower()
    # 에어컨의 온도를 30도로 설정합니다.
    my_air.temperature = 30
    # 에어컨의 온도를 출력합니다.
    my_air.print_temperature()
    # 에어컨의 온도를 높입니다.
    my_air.increase()
    # 에어컨의 온도를 낮춥니다.
    my_air.decrease()
    # 에어컨의 전원을 끕니다.
    my_air.togglePower()
```

## 실행결과

>>>

전원이 켜졌습니다.  
온도를 30도로 설정하였습니다.  
현재 온도는 30도 입니다.  
온도를 높입니다.  
현재 온도는 31도 입니다.  
온도를 낮춥니다.  
현재 온도는 30도 입니다.  
전원이 꺼졌습니다.



## ❖ 상속 (inheritance)

- 다른 누군가가 만든 기본 형태에 내가 원하는 것만 교체

## ❖ 다중 상속

- 다른 누군가가 만든 형태들을 조립하여 내가 원하는 것을 만드는 것

## ❖ 프로그래밍 언어에서 기반이 되는 것을 부모, 이를 기반으로 생성한 것을 자식이라 부름. 부모가 자식에게 자신의 기반을 물려주는 기능이므로 “상속”

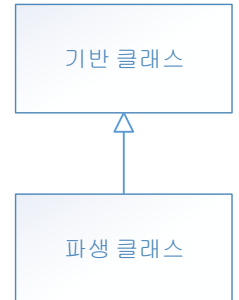


## ❖ “상속(Inheritance)”

- 한 클래스가 다른 클래스로부터 데이터 속성과 메소드를 물려받는 것.

```
class 기반 클래스:  
    # 멤버 정의
```

```
class 파생 클래스(기반 클래스)  
    # 아무 멤버를 정의하지 않아도 기반 클래스의 모든 것을 물려받아 갖게 됩니다.  
    # 단, 프라이빗 멤버(__로 시작되는 이름을 갖는 멤버)는 제외입니다.
```



## ❖ 실습

```
>>> class Base:  
        def base_method(self):  
            print("base_method")  
  
>>> class Derived(Base):  
        pass  
  
>>> base = Base()  
>>> base.base_method()  
base_method  
>>> derived = Derived()  
>>> derived.base_method()  
base_method
```

```
01 # 부모 클래스를 선언합니다.
02 class Parent:
03     def __init__(self):
04         self.value = "테스트"
05         print("Parent 클래스의 __init()__ 메소드가 호출되었습니다.")
06     def test(self):
07         print("Parent 클래스의 test() 메소드입니다.")
08
09 # 자식 클래스를 선언합니다.
10 class Child(Parent):
11     def __init__(self):
12         Parent.__init__(self)
13         print("Child 클래스의 __init()__ 메소드가 호출되었습니다.")
14
15 # 자식 클래스의 인스턴스를 생성하고 부모의 메소드를 호출합니다.
16 child = Child()
17 child.test()
18 print(child.value)
```

## 실행결과

```
Parent 클래스의 __init()__ 메소드가 호출되었습니다.
Child 클래스의 __init()__ 메소드가 호출되었습니다.
Parent 클래스의 test() 메소드입니다.
테스트
```



## ❖ 예외 클래스 만들기

- Exception 클래스 수정하여 CustomException 클래스 만들기

```
01 class CustomException(Exception):  
02     def __init__(self):  
03         Exception.__init__(self)  
04  
05 raise CustomException
```

실행결과

```
Traceback (most recent call last):  
  File "inherit02.py", line 5, in <module>  
    raise CustomException  
CustomException
```





- 예시 - 수정. 자식 클래스로써 부모의 함수 재정의하기

```
01 class CustomException(Exception):
02     def __init__(self):
03         Exception.__init__(self)
04         print("##### 내가 만든 오류가 생성되었어요! #####")
05     def __str__(self):
06         return "오류가 발생했어요"
07
08 raise CustomException
```

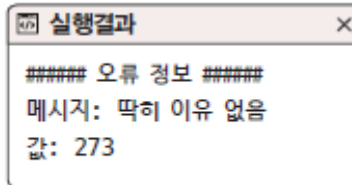
실행결과

```
##### 내가 만든 오류가 생성되었어요! #####
Traceback (most recent call last):
  File "inherit03.py", line 7, in <module>
    raise CustomException
CustomException: 오류가 발생했어요
```



- 예시 - 자식 클래스로써 부모에 없는 새로운 함수 정의하기

```
01  # 사용자 정의 예외를 생성합니다.
02  class CustomException(Exception):
03      def __init__(self, message, value):
04          Exception.__init__(self)
05          self.message = message
06          self.value = value
07
08      def __str__(self):
09          return self.message
10
11      def print(self):
12          print("##### 오류 정보 #####")
13          print("메시지:", self.message)
14          print("값:", self.value)
15  # 예외를 발생시켜 봅니다.
16  try:
17      raise CustomException("딱히 이유 없음", 273)
18  except CustomException as e:
19      e.print()
```



```
실행결과
##### 오류 정보 #####
메시지: 딱히 이유 없음
값: 273
```



### ❖ super()는 부모 클래스의 객체 역할을 하는 프록시(Proxy)를 반환하는 내장함수

- super() 함수의 반환 값을 상위클래스의 객체로 간주하고 코딩.
- 객체 내의 어떤 메소드에서든 부모 클래스에 정의되어 있는 버전의 메소드를 호출하고 싶으면 super()를 이용.

### ❖ 예제 : 09/super.py

```
class A:
    def __init__(self):
        print("A.__init__()")
        self.message = "Hello"

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

        super().__init__()
        print("self.message is " + self.message)

if __name__ == "__main__":
    b = B()
```

### • 실행 결과

```
>super.py
B.__init__()
A.__init__()
self.message is Hello
```



### ❖ 다중상속은 자식 하나가 여러 부모(?!)로부터 상속을 받는 것

- 파생 클래스의 정의에 기반 클래스의 이름을 콤마(,)로 구분해서 쪽 적어주면 다중상속이 이루어짐.

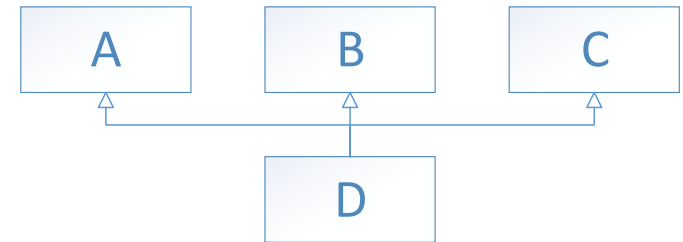
```
class A:  
    pass
```

```
class B:  
    pass
```

```
class C:  
    pass
```

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

클래스 D는 클래스 A, B, C를 부모로  
부터 상속받습니다.



## ❖ 다이아몬드 상속 : 다중 상속이 만들어 내는 곤란한 상황

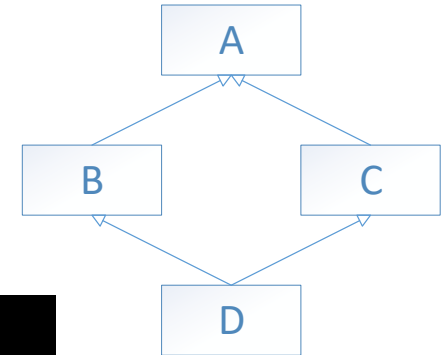
- D는 B와 C 중 누구의 method()를 물려받게 되는 걸까?

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

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

```
class C(A):  
    def method(self):  
        print("C")
```

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



```
>>> class A:  
        def method(self):  
            print("A")
```

```
>>> class B(A):  
        def method(self):  
            print("B")
```

```
>>> class C(A):  
        def method(self):  
            print("C")
```

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

```
>>> obj = D()  
>>> obj.method()
```

B

D는 B의 method()를 물려받았습니다.



### ❖ 메서드 탐색 순서 : Method Resolution Order, MRO

```
>>D.mro()  
[<class '__main__.D'>, <class '__main__.B'>, <class '__main__.C'>, <class '__main__.A'>, <class  
'object'>]
```

### ❖ class D(B, C)의 클래스 목록 중 왼쪽에서 오른쪽 순서

### ❖ 상속관계가 복잡하게 얹혀있다면 MRO를 살펴볼 것



## 상속 – 정적메소드와 클래스메소드의 차이

```
class Language:
    default_language = "English"

    def __init__(self):
        self.show = '나의 언어는 ' + self.default_language

    @classmethod
    def class_my_language(cls):
        return cls()

    @staticmethod
    def static_my_language():
        return Language()

    def print_language(self):
        print(self.show)

class KoreanLanguage(Language):
    default_language = "한국어"

if __name__ == "__main__":
    a = KoreanLanguage.static_my_language()
    b = KoreanLanguage.class_my_language()

    a.print_language()
    b.print_language()
```

**staticmethod**에서는 부모클래스의 클래스속성값을 가져오지만,

**classmethod**에서는 cls인자를 활용하여 cls의 클래스 속성을 가져옴



❖ OOP에서 오버라이딩의 뜻은 “기반(부모) 클래스로부터 상속받은 메소드를 다시 정의하다.”

### ❖ 실습 1

```
>>> class A:
    def method(self):
        print("A")
```

B는 A를 상속하지만, A의 method()를 물려받는 대신 자신만의 버전을 재정의(오버라이딩) 함.

```
>>> class B(A):
    def method(self):
        print("B")
```

```
>>> class C(A):
    def method(self):
        print("C")
```

C도 A를 상속하지만, A의 method()를 물려받는 대신 자신만의 버전을 재정의(오버라이딩) 함.

```
>>> A().method()
```

```
A
```

```
>>> B().method()
```

```
B
```

```
>>> C().method()
```

```
C
```





# 데코레이터 : 함수를 꾸미는 객체

## ❖ 데코레이터는 `__call__()` 메소드를 구현하는 클래스

- `__call__()` 메소드는 객체를 함수 호출 방식으로 사용하게 만드는 마법 메소드

```
>>> class Callable:  
    def __call__(self):  
        print("I am called.")
```

```
>>> obj = Callable()  
>>> obj()  
I am called.
```

인스턴스 뒤에 괄호 (와 )를 붙여 "호출" 하면, 내부적으로는 `__call__` 메소드가 호출됩니다.



# 데코레이터 : 함수를 꾸미는 객체

## ❖ 예제 : 09/decorator1.py(데코레이터 선언과 사용 1)

```
class MyDecorator:
    def __init__(self, f):
        print("Initializing MyDecorator...")
        self.func = f

    def __call__(self):
        print ("Begin :{0}".format( self.func.__name__))
        self.func()
        print ("End :{0}".format(self.func.__name__))
```

```
def print_hello():
    print("Hello.")
```

MyDecorator의 func 데이터 속성이 print\_hello를 받아둡니다.

```
print_hello = MyDecorator(print_hello)
```

```
print_hello()
```

MyDecorator의 인스턴스를 만들어지며 \_\_init\_\_() 메소드가 호출됩니다. print\_hello 식별자는 앞에서 정의한 함수가 아닌 MyDecorator의 객체입니다.

\_\_call\_\_() 메소드 덕에 MyDecorator 객체를 호출하듯 사용할 수 있습니다.

### • 실행 결과:

```
>decorator1.py
Initializing MyDecorator...
Begin :print_hello
Hello.
End :print_hello
```

# 데코레이터 : 함수를 꾸미는 객체

## ❖ 예제 : 09/decorator2.py (데코레이터 선언과 사용 2)

```
class MyDecorator:
    def __init__(self, f):
        print("Initializing MyDecorator...")
        self.func = f

    def __call__(self):
        print ("Begin :{0}".format( self.func.__name__))
        self.func()
        print ("End :{0}".format(self.func.__name__))

@MyDecorator
def print_hello():
    print("Hello.")

print_hello()
```

### • 실행 결과:

```
>decorator2.py
Initializing MyDecorator...
Begin :print_hello
Hello.
End :print_hello
```



# for 문으로 순회를 할 수 있는 객체 만들기

## - 이터레이터와 순회 가능한 객체

❖ 파이썬에서 for문을 실행할 때 가장먼저 하는 일은 순회하려는 객체의 `__iter__()` 메소드를 호출하는 것.

- `__iter__()` 메소드는 이터레이터(iterator)라고 하는 특별한 객체를 for 문에게 반환(이터레이터는 `__next__()` 메소드를 구현하는 객체)
- for문은 매 반복을 수행할 때마다 바로 이 `__next__()` 메소드를 호출하여 다음 요소를 얻어냄.

❖ `range()` 함수가 반환하는 객체도 순회가능한 객체.

❖ 실습 1

```
>>> iterator = range(3).__iter__()
>>> iterator.__next__()
0
>>> iterator.__next__()
1
>>> iterator.__next__()
2
>>> iterator.__next__()
Traceback (most recent call last):
  File "<pyshell#22>", line 1, in <module>
    iterator.__next__()
StopIteration
```

# for 문으로 순회를 할 수 있는 객체 만들기

## - 이터레이터와 순회 가능한 객체

### ❖ 예제 : 09/iterator.py(직접 구현한 range() 함수)

```
class MyRange:
    def __init__(self, start, end):
        self.current = start
        self.end = end

    def __iter__(self):
        return self

    def __next__(self):
        if self.current < self.end:
            current = self.current
            self.current += 1
            return current
        else:
            raise StopIteration()

for i in MyRange(0, 5):
    print(i)
```

#### • 실행 결과

```
0
1
2
3
4
```

## for 문으로 순회를 할 수 있는 객체 만들기 - 제네레이터

❖ 제네레이터(Generator)는 yield문을 이용하여 이터레이터보다 더 간단한 방법으로 순회가능한 객체를 만들게 해줌.

- yield문은 return문처럼 함수를 실행하다가 값을 반환하지만, return문과는 달리 함수를 종료시키지는 않고 중단시켜놓기만함.

❖ 예제 : 09/generator.py

```
def YourRange(start, end):  
    current = start  
    while current < end:  
        yield current  
        current += 1  
    return  
  
for i in YourRange(0, 5):  
    print(i)
```

- 실행 결과

```
0  
1  
2  
3  
4
```



## 상속의 조건 : 추상 기반 클래스

### ❖ 추상 기반 클래스(Abstract Base Class)는 자식 클래스가 갖춰야 할 특징(메소드)을 강제

- 추상 기반 클래스를 정의할 때는 다음과 같이 abc 모듈의 ABCMeta 클래스와 @abstractmethod 데코레이터를 이용

### ❖ 실습 1 (ABC의 규칙을 위반했을 때)

```
>>> from abc import ABCMeta
>>> from abc import abstractmethod
>>> class AbstractDuck(metaclass=ABCMeta):
    @abstractmethod
    def Quack(self):
        pass

>>> class Duck(AbstractDuck):
    pass

>>> duck = Duck()
Traceback (most recent call last):
  File "<pyshell#6>", line 1, in <module>
    duck = Duck()
TypeError: Can't instantiate abstract class Duck with abstract methods Quack
```

### ❖ 실습 2 (ABC의 규칙을 준수)

```
>>> class Duck(AbstractDuck):
    def Quack(self):
        print("[Duck] Quack")

>>> duck = Duck()
>>> duck.Quack()
[Duck] Quack
```





# Thank You !

파이썬 프로그래밍