

클래스의 추가적인 구문

목차

- 시작하기 전에
- 어떤 클래스의 인스턴스인지 확인하기
- 특수한 이름의 메소드
- 클래스 변수와 메소드
- 가비지 컬렉터
- 프라이빗 변수와 게터/세터
- 키워드로 정리하는 핵심 포인트
- 확인문제

[핵심 키워드] : isinstance(), 클래스 변수, 클래스 함수

[핵심 포인트]

클래스의 추가적인 기능에 대해 알아본다.

- 상속

- 어떤 클래스를 기반으로 그 속성과 기능을 물려받아 새로운 클래스 만드는 것

- `isinstance()` 함수

- 상속 관계에 따라서 객체가 어떤 클래스를 기반으로 만들었는지 확인할 수 있게 해주는 함수

- `str()` 함수

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
```

- 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.
```

- 다양한 보조 기능들
 - `__<이름>__()` 형태
 - 특수한 상황에 자동으로 호출되도록 만들어짐

— 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.


```
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 # 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

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
```

```
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()
```

```
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
```

- 클래스 변수
 - class 구문 바로 아래의 단계에 변수를 선언

```
class 클래스 이름:  
    클래스 변수 = 값
```

```
클래스 이름.변수 이름
```

— 활용 예시

```
01  # 클래스를 선언합니다.
02  class Student:
03      count = 0
04
05      def __init__(self, name, korean, math, english, science):
06          # 인스턴스 변수 초기화
07          self.name = name
08          self.korean = korean
09          self.math = math
10          self.english = english
11          self.science = science
12
13      # 클래스 변수 설정
```

실행결과

1번째 학생이 생성되었습니다.
2번째 학생이 생성되었습니다.
3번째 학생이 생성되었습니다.
4번째 학생이 생성되었습니다.
5번째 학생이 생성되었습니다.
6번째 학생이 생성되었습니다.

현재 생성된 총 학생 수는 6명입니다.

```
14     Student.count += 1
15     print("{}번째 학생이 생성되었습니다.".format(Student.count))
16
17     # 학생 리스트를 선언합니다.
18     students = [
19         Student("윤인성", 87, 98, 88, 95),
20         Student("연하진", 92, 98, 96, 98),
21         Student("구지연", 76, 96, 94, 90),
22         Student("나선주", 98, 92, 96, 92),
23         Student("윤아린", 95, 98, 98, 98),
24         Student("윤명월", 64, 88, 92, 92)
25     ]
26
27     # 출력합니다.
28     print()
29     print("현재 생성된 총 학생 수는 {}명입니다.".format(Student.count))
```

클래스 내부와 외부에서
클래스 변수에 접근할 때는
모두 `Student.count` 형태
(클래스이름.변수이름)를
사용합니다.

- 클래스 함수
 - 클래스가 가진 함수
 - '클래스가 가진 기능' 명시적으로 나타냄
 - **데코레이터** (decorator) : @classmethod

클래스 함수 만들기

```
class 클래스 이름:  
    @classmethod  
    def 클래스 함수(cls, 매개변수):  
        pass
```

클래스 함수 호출하기

```
클래스 이름.함수 이름(매개변수)
```


– 활용 예시 – Student.print()

```
01  # 클래스를 선언합니다.
02  class Student:
03      # 클래스 변수
04      count = 0
05      students = []
06
07      # 클래스 함수
08      @classmethod
09      def print(cls):
10          print("----- 학생 목록 -----")
11          print("이름\t총점\t평균")
12          for student in cls.students:
13              print(str(student))
14          print("-----")
15
```

→ Student.students라고 해도 상관없지만,
여기서는 매개변수로 받은 cls를 활용합니다.

```
16     # 인스턴스 함수
17     def __init__(self, name, korean, math, english, science):
18         self.name = name
19         self.korean = Korean
20         self.math = math
21         self.english = English
22         self.science = science
23         Student.count += 1
24         Student.students.append(self)
25
26     def get_sum(self):
27         return self.korean + self.math + \
28             self.english + self.science
29
30     def get_average(self):
```

```
31         return self.get_sum() / 4
32
33     def __str__(self):
34         return "{}\t{}\t{}".format(\
35             self.name,\
36             self.get_sum(),\
37             self.get_average())
38
39 # 학생 리스트를 선언합니다.
40 Student("윤인성", 87, 98, 88, 95)
41 Student("연하진", 92, 98, 96, 98)
42 Student("구지연", 76, 96, 94, 90)
43 Student("나선주", 98, 92, 96, 92)
44 Student("윤아린", 95, 98, 98, 98)
45 Student("윤명월", 64, 88, 92, 92)
46 Student("김미화", 82, 86, 98, 88)
47 Student("김연화", 88, 74, 78, 92)
48 Student("박아현", 97, 92, 88, 95)
49 Student("서준서", 45, 52, 72, 78)
50
51 # 현재 생성된 학생을 모두 출력합니다.
52 Student.print()
```

----- 학생 목록 -----		
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
김미화	354	88.5
김연화	332	83.0
박아현	372	93.0
서준서	247	61.75

- 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")
```

실행결과

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

- 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

- 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
```

```
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'
```

- 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
```



```
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
```

- 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
```

- 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.")
```

```
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.
```

- 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"

```

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
```

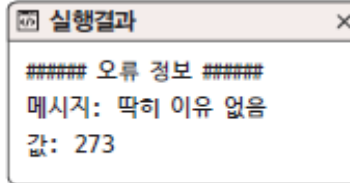
실행결과

```
##### 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()
    
```



```

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

- **isinstance()** : 어떤 클래스의 인스턴스인지 확인할 때 사용하는 함수
- **클래스 변수, 클래스 함수** : 클래스 이름 뒤에 마침표 찍고 바로 사용할 수 있는 클래스가 갖는 변수와 함수
- **상속** : 어떤 클래스 기반으로 그 속성과 기능을 물려받아 새로운 클래스 만드는 것

- 슬라이드 #10 코드의 compare_func.py를 수정해서 Student 객체를 숫자와 비교했을 때 학생의 성적 평균과 비교가 일어나게 해보세요.
 - 예를 들어 다음과 같습니다.

```
test = Student("A", 90, 90, 90, 90)
print(test == 90)           # → True
print(test != 90)          # → False
print(test > 90)            # → False
print(test >= 90)          # → True
print(test < 90)           # → False
print(test <= 90)          # → True
```

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

    def get_sum(self):
        return self.korean + self.math + \
            self.english + self.science

    def get_average(self):
        return self.get_sum() / 4

    def __str__(self, value):
        return self.get_average()

    def __repr__(self, value):
        return self.get_average()

    def __str__(self, value):
        return self.get_average()

    def __repr__(self, value):
        return self.get_average()

    def __str__(self, value):
        return self.get_average()

    def __repr__(self, value):
        return self.get_average()
```

```
# 학생을 선언합니다.
```

```
test = Student("A", 90, 90, 90, 90)
```

```
# 출력합니다.
```

```
print("test == 90:", test == 90)
```

```
print("test != 90:", test != 90)
```

```
print("test > 90:", test > 90)
```

```
print("test >= 90:", test >= 90)
```

```
print("test < 90:", test < 90)
```

```
print("test <= 90:", test <= 90)
```