class 상속 그리고 예외처리

자료	Python
구분	Python

[교재내용]

- 상속
- 에러 예외처리

23.07.27
라이브 수업 이후 교육생의 이해를 위해서 만든 파일

```
'''
오늘 공부하는 내용은 상속
그리고 oop 에서의 중요한 개념인 상속 추상화 캡슐화 다향성 에 대해서
함께 살펴 볼 것이다.

상속: 부모/자식관계의 클래스 -> 부모로 부터 속성과 메소드를 물려받아 사용
     (SUPER를 통해서 부모의 요소를 호출가능)
추상화: 복잡한 것은 숨기고 필요한 것은 나타냄
       (자식들의 공통사항은 부모클레스에 구현한다는 내용)
캡슐화: 객체안의 코드를 외부로 부터 집접적인 접근 차단
       ( GETTER SETTER )
다형성: 동일한 메서드가 클래스에 따라 다르게 행동할 수 있음
       (메서드 오버라이딩)
'''

# 객체지향의 핵심개념으로 (상속 추상화 캡슐화 다향성) 이 있다.

# 상속을 해보자
# 상속은 물려받다 라는 뜻으로 재산을 상속받다와 같은 의미다.
# 클래스를 하나 더 만드는데 원래 있던 클래스의 기능을 그대로 물려 받는 것이다.

# 기존에 있던 클래스 (상위,부모클래스)의 모든 속성과 메소드를
# 새로 만든 클래스 (하위,자식클래스)가 그대로 물려 받음으로써
# 코드 재사용성이 높아짐


# 다시 한번 말하지만 기존에 있는 클래스를 변경하지 않고 새로운 클래스에
# 기능을 추가하거나 변경할때 사용한다.

# 어제 만들어서 사용했던 plus_minus클래스를 또 사용해 보자.
class plus_minus:
    def __init__(self,first,second):
        self.first=first
        self.second=second
    def plus(self):
        result=self.first+self.second
        return result
```

```
    def minus(self):
        result=self.first-self.second
        return result

# 위의 클래스는 숫자 2개의 합과 차를 반환하는 클래스 이다.

# 하위 클래스를 하나 만들 것인데
# 숫자 3개의 곱을 구해주는 하위클래스를 만들꺼임 (자식클래스를 만들 것임)
# 부모클래스의 속성과 매소드 들이 자식클래스로 상속이 된다고 했음!!

class morefunction(plus_minus):
    def __init__(self, first, second,third):
        super().__init__(first, second) # super는 부모클래스의 init메소드를 그대로 호출하는것
        self.third=third                # third만 추가

    def mul(self):
        result=self.first*self.second*self.third
        return result

b=morefunction(3,4,5)
#부모 클래스의 매서드 호출도 가능
print(b.mul())
print(b.plus())

# 단, 자식클래스에서 생성된 메서드는 부모 클래스에서 사용 불가함을 주의 !!


# 메서드 오버라이딩
# 오버라이딩이란 덮어쓰기를 말한다.
# 부모클래스에 있는 메서드!!! 를 동일한 이름으로 다시 만드는 것이다.
# (부모 클래스의 메서드 이름과 기본 기능은 그대로 사용하지만, 특정 기능을 바꾸고 싶을 때 사용)

# 이번에는 아래에 코드를 구현 해 볼 것인데 무엇을 구현 할 것이냐면
# 부모 클래스에서 상속받은 plus의 메소드의 기능에서
# 숫자 2개가 아닌 숫자 3개의 합이 100이 넘는다면
# '버그' 라고 출력이 되도록
# 자식클래스에서 plus의 메소드를 업그레이드 하려 한다고 가정하자


class morefunction(plus_minus):
    def __init__(self, first, second,third):
        super().__init__(first, second) # super는 부모클래스의 init메소드를 그대로 호출하는것
        self.third=third
    def mul(self):
        result=self.first*self.second*self.third
        return result
    def plus(self):
        get_sum=self.first+self.second+self.third
        if get_sum>100:
            print('버그')
        else:
            return print(get_sum)
a=morefunction(1,1,99)
a.plus()

# 이처럼 부모클래스의 plus 메서드를 자식클래스에서
# 새롭게 다시 정의하는 것을 메서드 오버라이딩 이라고 한다.
# 이번에는 자식클래스에서 plus 메서드를 다시 정의 했지만 (메서드 오버라이딩을 했지만)
```

```
# 부모 클래스에서의 plus 메서드를 자식클래스에서도 또 사용하고 싶다면!!
# 이때 super() 를 이용해서 사용 할 수 있다.


class morefunction(plus_minus):
    def __init__(self, first, second,third):
        super().__init__(first, second) # super는 부모클래스의 init메소드를 그대로 호출하는것
        self.third=third

    def mul(self):
        result=self.first*self.second*self.third
        return result

    def plus(self):
        get_sum=self.first+self.second+self.third
        if get_sum>100:
            print('버그')
        else:
            return print(get_sum)

    def parents_plus(self):
        ret=super().plus()    # 부모 메서드에서의 plus메서드 호출시 super 활용 가능
        return ret

t=morefunction(500,200,400)
print(t.parents_plus())       # 자식이 아닌, "부모 클래스"에서의 plus 메서드를 사용

# 부모 클래스에서의 plus 메서드를 사용하면 파라매터가 2개 임으로
 # 500+200인 700이 출력됨

# 만약에 상속이 복잡해서 상속 구조(부모자식 관계)를 확인하고 싶다면
print(morefunction.mro())   # method resulution order
```

이번에는 다중상속에 대해서 알아보자.

다중 상속

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

    def greeting(self):
        return f'안녕, {self.name}'


class Mom(Person):
    gene = 'XX'

    def swim(self):
        return '엄마가 수영'


class Dad(Person):
    gene = 'XY'

    def walk(self):
        return '아빠가 걷기'
```

```python
class FirstChild(Dad, Mom):
    def swim(self):
        return '첫째가 수영'

    def cry(self):
        return '첫째가 응애'


baby1 = FirstChild('아가')
print(baby1.cry()) # 첫째가 응애
print(baby1.swim()) # 첫째가 수영
print(baby1.walk()) # 아빠가 걷기
print(baby1.gene) # XY
```

90

```python
# 부모클래스를 두개 이상의 클래스를 상속받는 경우
# 하나의 부모에 두개 이상의 자식으 있는 경우의 코드를 살펴 볼 것이다.
# 어떻게 출력이 되는지 잘 살펴보자.

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

    def greeting(self):
        return f'안녕, {self.name}'

class Mom(Person):
    gene = 'XX'

    def swim(self):
        return '엄마가 수영'

class Dad(Person):
    gene = 'XY'

    def walk(self):
        return '아빠가 걷기'

class FirstChild(Dad, Mom):
    def swim(self):
        return '첫째가 수영'

    def cry(self):
        return '첫째가 응애'

baby1 = FirstChild('아가')  # 부모의 부모(person에 의하여) baby1.name은 아가가 됨
print(baby1.cry()) #첫째가응애
print(baby1.swim()) #첫째가수영 -> swim이 fisrtchild에서 다시 정의가 됨!!
print(baby1.walk()) #아빠가걷기 -> firstchild에서 정의되지 않아서 부모클래스에서 가져옴
print(baby1.gene) #XY -> dad가 먼저 상속되었기 때문에 진 에는 XY가 뜬다.
```

```
                    # -> 중복된 속성이나 메서드는 상속 순서에 의해 결정됨
###################################

# 추상화
# 공통적으로 들어가는 속성과 메서드는 상위클래스에서 구현 해 주고
# 하위객체에는 그 객체의 고유기능만 추가하는 것을 이론적으로 추상화라 한다.

# 현대차 쌍용차 볼보 아우디 등의 객체가 있다.
# 위 자동차 들의 공통의 속성이나 기능을 묶어 상위 클래스에 구현하고
# 하위객체에는 각각 객체의 고유기능만 추가하는 것을 추상화라 한다.

# 추상화를 통해 불필요 하게 반복되는 코드를 줄이고 하위객체에 필요한 정보만 표현하면서
# 프로그래밍을 하는 것을 말한다.

###################################

# 다형성 : 동일한 메서드가 클래스에 따라 다르게 행동할 수 있음.
# 메서드 오버라이딩이 대표적인 예다.
# 상속받은 클래스에서 같은 이름으로 매서드를 덮어 씌울 수 도 있고
# 부모 클래스의 매서드를 사용하고 싶으면 super를 사용하기도 했다.
# 이처럼 동일한 메서드가 클래스에 따라 다르게 행동할 수 있다는 개념이 다형성이다.

###################################

# 캡슐화
# 객체의 일부 구현 내용을 외부에서 사용하지 못하게 차단하는 것
# 언제사용? 클래스 사용자가 클래스를 만든 개발자의 의도를 벗어나
# 사용하는 것을 막기위해 접근을 제한함

# 개발자가 새로운 모듈을 만들어서 오픈소스로 공개를 하였다고 가정하자.
# 많은 사람들은 새로 만들어진 모듈의 메서드를 사용 해 보기 시작했다.
# 그런데 유저들은 모듈을 만든 개발자가 적어놓은 사용법 또는 README 파일, 공식문서등을
# 통해서 사용법을 명시해 놓아도 유저들은 가이드를 한 사용 방법대로 사용하지 않는것이 유저들이다.
# 그럼 개발자의 의도를 벗어나 사용하는 것을 막기위해서, 즉
# 특정 클래스 매서드 들의 접근을 막을때 사용한다.

# 캡슐화를 통해 클래스 내 특정 코드를 외부로 부터의 접근을 제어하는데 3가지 가 있다.
# PUBLIC # PROTECTED # PRIVATE

# 아래 코드에 NAME 이라는 속성을 사용할 것인데. 각각의 케이스 별로 어떠한 기능이 있는지
# 결론부터 보자.

#Public     :   name  아래 코드에는 이렇게 사용할 것이며 외부로부터 모든 접근 허용한다는 뜻
#Protected  :  _name  자기자신 클래스 내부 혹은 상속받은 자식 클래스에서만 접근 허용하겠다.
#Private    : __name  자기자신 클래스 내부의 메서드에서만 접근 허용 (자식클래스 x)

#하나씩 예를 살펴보자

# public
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = 21

p1 = Person('김영일', 21)

print(p1.name) # 김영일
p1.age=40
```

```
print(p1.age) # 김영일

# Person 클래스의 인스턴스인 p1은 이름(name)과 나이(age) 모두 접근 가능합니다.
# PUBLIC 이니까 ㅎㅎ


# Protected
# "암묵적 규칙"에 의해 메소드를 호출 시 부모클래스 내부나 자식클래스에서만 호출이 가능하다
# 그러나 강제성은 없으므로 실제로는 Public과 거의 동일하게 외부 접근가능하다.
# (정말 막아주지는 않고 암묵적인 규칙)

# 따라서 속성에 언더바 하나 있으면.. 이 뜻은 '
# 해상 속성은 부모클래스나 자식클래스 에서만 호출하라는 개발자들 끼리의
# 암묵적인 약속이라고 보면 된다.
# 그래서 실제로는 PUBLIC과 마찬가지로 외부에서 호출해도 다 잘 작동됨
  ~~~(크게 의미가 없다는..(내생각))~~~

class Person:
    def __init__(self, name, age):
        self.name = name
        self._age = age
    def getage(self):
        return self._age

p1 = Person('조수훈', 21)
print(p1.getage()) # 21
print(p1._age) # 21


# private
# private은 자기 클래스 내부의 메서드에서만 접근을 허용
# 예제를 보자.

#private

class Person:
    def __init__(self,name,age):
        self.name=name
        self.age=age
    def getage(self):
        return self.age

p2=Person('이동준',50)
print(p2.name)  # 이동준
print(p2.age)   # 50
print(p2.getage())  # 50

# 위의 코드는 pubulic 으로 클래스 밖 인스턴스에서 접근 가능하지만
# 아래 코드는 age 속성을 언더바 2개(__)를 통해서 private으로 바꿔 보았다.

class Person:
    def __init__(self,name,age):
        self.name=name
        self.__age=age      #언더스코어 2개를 붙여
    def getage(self):
        return self.__age   #언더스코어 2개를 붙여

p2=Person('이동준',50)
```

```
print(p2.name)
#print(p2.__age)
print(p2.getage())
# 이동준
# 버그!!    클래스 밖 인스턴스로 직접 접근 못함!! private 비공개 속성임
# 50         함수를 호출하면 함수 내부에서 self.__age속성값으로 접근 후 값을 반환


# 이렇게 언더스코어(__) 2개를 통해
# 클래스 멤버 속성을 외부에 있는 인스턴스로직접 접근을 제한했습니다.
# 그래서 print(p2.__age) 라고 하면 속성을 직접적으로 출력이 안된 것이다.

# 하지만 외부에서도 값을 변경할 수 있도록 하고 싶다면?
# getter 메소드와 setter 메소드를 만들어서 사용하면 가능 하다.
# getter 메소드와 setter 메소드 는 각각 @property 그리고 @setter라는 데코레이터를 사용한다.

# getter 메소드는 private한 변수 값을 읽을 목적으로 만들 것이고
# setter 메소드는 private한 변수 값을 변경할 목적으로 만들 것이다.


# 먼저 @property 그리고 @setter 데코레이터를 사용하지 않고
# getter / setter 메소드를 직접 구현해서 private한 변수값을
# 확인하고 변경하는 코드를 한번 살펴보자.

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

    def getter(self):          # private한 속성값을 "확인"하고자 함 (getter함수가 됨)
        return self.__age

    def setter(self,value):    # private한 속성값을 "변경"하고자 함 (setter함수)
        self.__age=value

k=Person('kevin',30)
print(k.getter())
k.setter(60)
print(k.getter())

# 위와 같이 클래스의 속성값을 getter와 setter를 이용해서 확인 또는 변경이 가능한데
# 위 코드를 더 간단하게 그리고 가독성 좋게 사용하기 위해서 데코레이터를 사용한 코드를 보자
# 어제 데코레이터가 무엇인지 봤으니까 또 설명은 안할께요!

# getter 메소드에는 @property라는 데코레이터를 사용하고
# setter 메소드에는 @변수.setter 로 데코레이터를 사용할 것이다.

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

    @property                  # getter 함수 위에는 @property라고 적어주고
    def age(self):             # 비공개 속성의 값을 확인하기 위한 용도
        return self.__age

    @age.setter             # @메서드명.setter라고 적어준다
```

```
    def age(self,value):    # 비공개 속성의 값을 변경하기 위한 용도
        self.__age=value

k=Person('kevin',40)
print(k.age)     # getter
k.age=20         # setter
print(k.age)     # getter

# 참고로 @property 데코레이터 사용을 하면 메서드 호출시 소괄호 () 를 생략해야 한다.
# 관례상 setter getter 함수명은 변수명을 따른다.

여기까지 class 에 대해서 살펴 보았다. 알면 쉽고 모르면 어렵다. ^^;;;;


# 에러와 예외처리
# 잘못된 문법으로 인하여 어떤 에러 메시지가 뜨는지 확인해 보는 시간이다.

# if True:
#     print('참')
# else
#     print('거짓')


# : 누락으로 문법 에러
# SyntaxError: invalid syntax


# ' 누락으로 문법 에러
# print('hi)
# SyntaxError: EOL while scanning string literal



# 예외 - 문법적으로 맞음 그런데 실행하면 발행하는 에러

# 10 * (1/0)
# ZeroDivisionError: division by zero
# 파이썬에서 어떤수를 0으로 나누면 에러남


# print(abc)
# NameError: name 'abc' is not defined
# 선언하지 않은 변수의 값을 출력하고자 함


# 1 + '1'
# TypeError: unsupported operand type(s) for +: 'int' and 'str'
# 타입이 다른 자료형의 합은 지원하지 않음


# round('3.5')
# TypeError: type str doesn't define __round__ method
# 실수 가 아닌 문자열을 반올림 할 수 없음


# import random
# random.sample([1, 2, 3])
# TypeError: sample() missing 1 required positional argument: 'k'
```

```
# 랜덤 클래스 내 sample 메서드 호출시 매개변수 누락됨

# random.choice([1, 2, 3], 6)
# TypeError: choice() takes 2 positional arguments but 3 were given
# 매개변수가 2개여야 하는데 매개변수 개수가 초과됨

# int('3.5')
# ValueError: invalid literal for int() with base 10: '3.5'
# int()는 정수가 아닌 값을 받았을 경우 에러가 발생


# numbers = [1, 2]
# numbers.index(3)
# ValueError: 3 is not in list
# 존재하지 않는 값을 찾을 경우

# empty_list = []
# empty_list[-1]
# IndexError: list index out of range
# 없는 인덱스를 조회할 경우

# songs = {'new jeans': 'supershy'}
# songs['ive']
# KeyError: 'ive'
# 없는 키로 접근할 경우

# import reque
# ModuleNotFoundError: No module named 'reque'
# 존재하지 없는 모듈을 import 할 경우


# for i in range(3):
# print(i)
# IndentationError: expected an indented block
# 들여쓰기 잘못했을 경우


# 예외처리

# try:
#     코드작성
# except (예외):
#     코드작성


# try:
#     num = input('값을 입력하시오 : ')
#     print(int(num))
# except ValueError:
#     print('숫자를 입력하라니까!!')

# 만약에 ssafy라고 입력시
# 숫자를 입력하라니까 가 출력되겠죠?
```

```
# except 문은 try 문에서 예외 발생시 실행

# try:
#     num = input('값을 입력하시오: ')
#     100/int(num)
# except ValueError:
#     print('숫자를 넣어')
# except ZeroDivisionError:
#     print('0으로 나눌 수 없어')
# except:
#     print('모르겠지만 오류야!')




# try:
#     age=int(input('나이를 입력하세요: '))
# except:
#     print('입력이 정확하지 않습니다.')
# else:
#     if age <= 18:
#         print('미성년자는 출입금지입니다.')
#     else:
#         print('환영합니다.')
# finally:
#     print('수고했어요')


# try문 수행중 오류가 발생하면 except절이 수행되고 오류가 없으면 else절이 수행된다.
# finally절은 try문 수행 도중 예외 발생 여부에 상관없이 항상 수행

# ~~~(알고리즘 문제 푸실 때는 try except 사용을 추천하고 싶지 않습니다. !! )~~~
```