파이썬 활용 및 OOP

소요시간 : 13 min

> 파이썬에서 활용될 수 있는 내용과 OOP로 인해 발생되는 부분을 지속적으로 알아봅니다.

파이썬 활용 및 OOP내용을 익힌다.

데이터 캡슐화와 접근제어

Name	Notation	Behavior
name	Public	Can be accessed from inside and outside
_name	Protected	Like a public member, but they shouldn't be directly accessed from outside
__name	Private	Can't be seen and accessed from outside

💡 생각하는 시간 : 캡슐화를 하는 이유

- 캡슐화 : object 및 소스코드 **구현에 대한 상세정보를 분리**하는 과정이다.
- **모듈화**가 가능해진다.(함수, 메소드, 클래스 등을 활용한 기능 분리.)
- 기능이 분리되어있으니 **디버깅**을 하는 경우 편리하다.
- 프로그램이 기능별로 분리되어있으니 **소스코드의 목적**을 알기쉽다.
- 파이썬은 object 접근제어를 위한 **접근제어자를 제공하지 않기때문에** 변수, 메소드, 함수에 직접 접근할 수 있다.
    - 파이썬에서는 상단 표와 같이 **직접 접근을 허용하지 않는 규칙**을 만들었다.
    - Notation : 접근 정도를 나타내는 명칭이며 Private -> Protected -> Public 순서로

외부에서 무분별한 접근을 막기위해 위와 같은 개념이 생겨났다.

- 외부 object가 속성이나 메소드에 대한 접근을 막기 위해 **이중 밑줄 __을 접두사**로 추가할 수 있다.
- '**_클래스이름__메소드이름**' 형태로 이름을 변환시켜, 부모클래스와 서브클래스의 변수나 메소드이름을 구분짓는다.

```
class Point:
   def __init__(self, x, y):
       self.x = x
       self.y = y
       self.__private_name = "private 접근"

class Point_sub(Point):
  def __init__(self,x, y):
    Point.__init__(self,x,y)
    self.x = x
    self.y = y

  def __sub(self):
    self.__x = 10
    self.__y = 20
  def sub(self):
    self.__sub()
    print(self.__x)
    print(self.__y)

my_point = Point(1, 2)

my_point_sub = Point_sub(10,20)


 # case 1 - error case
#print(my_point.__private_name)

 # case 2
#print(my_point._Point__private_name)    # 를

 # case 3
my_point_sub.sub()    # 변환된 이름에 대해 값
```

- 따라서, 코드상에서 접근을 강제로 막을 방법은 없다.

```
class A(object):
  def __init__(self):
    self.__X = 3        # self._A__X 로 변환
  def __test(self):     # _A__test() 로 변환
    print('self.__X' ,self.__X)
  def bast(self):
    self.__test()

class B(A):
  def __init__(self):
    A.__init__(self)
    self.__X = 20       # self._B__X 로 변환
  def __test(self):     # _B__test() 로 변환
    print(self.__X)
  def bast(self):
    self.__test()

a = A()
#a.__test()
a._A__test()      # 변형된 이름을 통해 접근가능
#a.bast()

b = B()
b._B__test()      # 변형된 이름을 통해 접근가능
b.bast()
dir(b)            # 상속을 받았기 때문에 B클래스
b._A__X
```

- 언더스코어(_) 참고자료 : 파이썬 언더스코어

🔍 자세히보기 : 속성(attribute)충돌

- 위에서 설명한 _와 __의 활용에 대해 생각해보자.
- 프로그램이 길어지고 다양한 변수를 선언하는 경우 클래스의 속성이 충돌(변수의 중복)할 수 있다.

```
class parent_class:
  def __init__(self):
    self.value = 30
    self._value = 40
    self.__value = 50

  def get(self):
    return self.value

class sub_class(parent_class):
  def __init__(self):
    super().__init__()
    self.__value = 20    # 위의 parent_class

s = sub_class()
print(s.value) # public
print(s._value)# protected
#print(s.__value)
#print('parent_class value:',s.get(),' sub_c
```

- 위처럼 속성의 충돌(같은 값)이 발생하는 이유는 대체로 프로그램에서 중복되는 속성(attribute)을 활용하는 경우이다.
- 속성명을 다르게 해줘도 되지만 파이썬에서 활용할 수 있는 '**비공개 속성**' 을 활용할 수 있다.
- 비공개 속성은 위에서 설명한 '__'를 활용한다.

```
class parent_class:
  def __init__(self):
    self.__value = 10   # parent_class는 '__'

  def get(self):
    return self.__value # parent_class는 '__'

class sub_class(parent_class):
  def __init__(self):
    super().__init__()  # parent_class 호출을
    self._value = 20    # sub_class는 '_'


s = sub_class()
print('parent_class value:',s.get(),' sub_cl
```

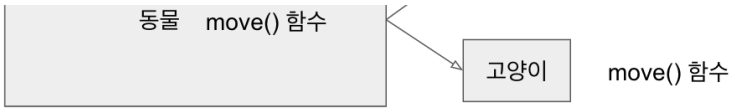
🔍 자세히보기 : 파이썬에서는 getter,setter에 대해서 명시적으로 표현하지 않기 때문에 property 키워드를 사용한다.

- 따라서 내장함수와 데코레이터를 활용하여 코드를 간결화하고 기존 인스턴스 속성에 새로운 기능을 제공할 수 있다.
    - 파이썬의 **내장함수 property()** : getter/setter에 대해서 **변수**처럼 활용할 수 있게 해준다.
    - **데코레이터로서의 @property** : 기존의 메소드에 대해서 새로운 기능을 제공할 수 있다.

      (단, 데코레이터를 적용한 메소드를 재사용할 수 없다.)

```
class Color:
    def __init__(self, rgb_value, name):
        self.rgb_value = rgb_value
        self._name = name
    def _set_name(self, name):
        if not name:
            raise Exception("Invalid Name")
        self._name = name

    def _get_name(self):
        return self._name
    name = property(_get_name, _set_name)

c = Color("#0000ff", "bright red")
print(c.name)   # property를 통해 변수처럼 get

c.name = "red"   # property를 통해 변수처럼 se
print(c.name)


# case 2 - @property 활용

class Color:
    def __init__(self, rgb_value, name):
        self.rgb_value = rgb_value
        self._name = name
    #  def _set_name(self, name):
    #      if not name:
    #          raise Exception("Invalid Name
    #      self._name = name

    @property                # 데코레이터 pro
    def _get_name(self):
        return self._name

c = Color("#0000ff", "bright red")
print(c._name)
# print(c.name)   # 이 경우는 에러이다. 왜 일까

c.name = "red"
print(c.name)
```

메소드 오버라이딩



생각하는 시간 : 메소드 오버라이딩은 상속개념을 기반으로 한 개념이다.

- 오버라이딩은 **우선시하다**라는 의미로, 상속처럼 부모클래스의 메소드를 재호출하는 것이 아니라 **같은 이름의 메소드를 신규 생성**하는 것이다.
- 새로운 이름의 메소드를 기능별로 만들면 되는데 부모클래스의 메소드이름을 그대로 사용하는 이유는 무엇일까?
    - **중복되는 기능(메소드)은 기존 부모클래스의 메소드(기능)로 재사용**하고, 다르게 사용하려면 **재정의**하는 개념으로 활용할 수 있다.
        - 프로그래밍의 핵심이 되는 **재사용**을 위함이다.
        - 메소드 오버라이딩도 다형성 개념의 한 종류이다.

```
class Bicycle():
     def exclaim(self):
       print("부모클래스 자전거")

class Specialized(Bicycle):             # 부모
    def exclaim(self, specialized):     # 메소
         print("자식클래스 재정의",specialized
         return 'return 자식클래스 재정의'+spe


a_bike = Bicycle()
a_specialized = Specialized()

# 출력1 - 부모클래스 메소드
a_bike.exclaim()

# 출력2 - 오버라이딩된 자식클래스 메소드(파라미터
a_specialized.exclaim('specialized test')
```

```
        print("부모클래스 자전거")

class Specialized(Bicycle): # 부모클래스를 상
    def exclaim(self):    # 메소드 오버라이딩
        super().exclaim() # super는 위의 부모
        print('super를 활용한 부모클래스의 메소

a_bike = Bicycle()
a_specialized = Specialized()

# 출력1 - 부모클래스 메소드
a_bike.exclaim()

# 출력2 - super : 부모클래스 메소드 그대로 활용
a_specialized.exclaim()
```

super 사용하기

- 아래 예시에서 Graduate(자식클래스)는 Student(부모클래스)가 가지고 있는 모든 **매개변수(파라미터, arguments)**를 사용한다.
    - 상속을 통한 재사용을 하는 경우, 아래와 같이 **다른 매개변수(graduation_date)도 신규 생성**할 수 있다.

```
            self.name = name
            print(self.name)

    class Graduate(Student):    # 부모클래스 상속받
        def __init__(self, name, graduation_date
            super().__init__(name)   # super를 
            self.graduation_date = graduation_da

    a_student = Student('stu_class')
    a_graduate = Graduate('gradu_class',11)

    a_student.__init__('stu')
    a_graduate.__init__('gradu',11)

    # graduate 인스턴스 생성
    a_graduate.name             #
    a_graduate.graduation_date  #
```

🔍 자세히보기 : 리스트 반복의 경우 range와 enumerate 내장함수를 비교하면서 사용해보는게 좋다.

```
    # range보다는 enumerate를 사용한다.
    # range 내장 함수는 정수집합을 반복하는 상황에서

    # 하지만 리스트에 대해서 특정 인덱스에 따라 반복을
    # 인덱스를 사용해 배열원소에 접근해야 한다.(index

    # case 1 - range를 통한 반복
    fla_list = ['A','B','C','D']

    len_list = len(fla_list)            # len을 
    index_list = fla_list.index('B')    # index를

    for i in range(0, len_list):
      print('range with len method : ',i)

    print('index method : ',index_list)
```

```
# case 2 - enumerate 활용
fla_list = ['A','B','C','D']
enu_list = enumerate(fla_list)
# print(enu_list)
# print(next(enu_list))
# print(next(enu_list))

# enumerate는 인덱스와 값을 함께 출력해주기 때문

for i,enu_list in enumerate(fla_list):
  print(f'{i+1}: {enu_list}')


# 위의 방법을 range로 해보고 비교해보자.

for i in range(0,len(fla_list)):
  fla = fla_list[i]
  print(f'{i+1}: {fla}')
```

Mark as Completed

< Prev Next >