

Professional Coding Specialist

COS Pro 파이썬 1 급

4 강. 문법 정리 4

1. 클래스
 2. 람다, 아이터레이터, 제너레이터,
추상 클래스, 뮤터블
-

과정 소개

COS Pro 1 급 파이썬 시험 대비를 위한 문법 정리 네 번째 시간으로 객체 지향 언어의 특징인 클래스의 개념에 대해 정리하고 파이썬에서 클래스를 정의하고 객체를 생성하여 활용하는 법을 배운다. 그 밖에 기타 사항으로 람다, 아이터레이터, 제너레이터, 추상 클래스의 기능과 사용하는 방법을 알아보고 mutable/immutable 의 개념과 차이점을 이해하여 COS Pro 1 급 파이썬 문제 풀이에 활용할 수 있도록 한다.

학습 목차

1. 클래스
2. 람다, 아이터레이터, 제너레이터, 추상 클래스, 뮤터블

학습 목표

1. 클래스와 인스턴스의 개념을 이해하고 차이점을 설명할 수 있다.
2. 파이썬 클래스를 정의하고 클래스를 기반으로 하는 인스턴스를 생성하여 활용하는 프로그래밍을 할 수 있다.
3. 클래스 상속을 이용하여 클래스를 정의할 수 있다.
4. 한 줄짜리 함수인 람다 함수를 정의해서 사용할 수 있다.
5. iterable 객체와 iterator 의 의미를 이해하고 설명할 수 있다.
6. 추상 클래스의 의미와 기능을 이해하고 이를 활용하여 프로그래밍 할 수 있다.
7. mutable/immutable 객체의 차이를 알고 이를 구분할 수 있다.

1

클래스

1. 객체지향(Object Oriented) 프로그래밍 언어


1) 특징

- 프로그램을 명령어의 목록으로 보는 시각에서 벗어나 여러 개의 독립된 객체들의 모임으로 구성된 것으로 인식.
- 각각의 객체는 메시지를 주고 받고, 데이터 처리.


2) '클래스'란?

클래스(Class)	특정 객체를 생성하기 위해 속성(변수)과 기능(메소드)을 정의하는 틀의 역할을 하는 것. 자동차 생산을 예로 들면, 자동차를 만들기 위한 틀(거푸집) 같은 역할을 하는 존재.
객체(Object)	클래스를 기반으로 만들어진 실제 사물. 자동차 거푸집을 이용해서 생산된 자동차와 같은 것.

예 1) 자동차 클래스를 기반으로 하여 빨강색 차와 노란색 차 객체 생성.

자동차 클래스		객체로 생성된 자동차
속성(Attribute)	기능(method)	
모델명	달린다()	
가격	멈춘다()	
색상		

예 2) 버튼 클래스를 기반으로 확인 버튼과 취소 버튼 객체 생성.

버튼 클래스		객체로 생성된 버튼
속성(Attribute)	기능(method)	
size	click()	
color	double_click()	
shape		

- ➔ 공통된 속성과 기능에 모아서 클래스를 정의한 뒤 필요에 따라 객체를 생성해서 사용하면 공통적으로 필요한 프로그램 코드를 중복 작성할 필요가 없음.

2. 파이썬 클래스

1) 클래스 정의.

```
class 클래스 이름:

    멤버 변수

    def __init__(self, 매개변수들): 초기화
        ... (Initializer)

    def __del__(self, 매개변수들): 소멸자
        ...

    def 메소드이름(self, 매개변수들): 메소드
```

- **멤버 변수** 객체의 속성, 데이터
- **초기화(생성자)** 객체가 생성될 때 실행되는 special method. 주로 멤버 변수의 초기화.
- **소멸자** 객체가 소멸될 때 실행되는 special method.
- **메소드** 객체의 기능, 행동, 함수

2) 인스턴스(Instance) : 클래스를 기반으로 실제 생성된 객체

- | | |
|------------------|-----------------------|
| • 인스턴스 생성 | 인스턴스 변수 이름 = 클래스 이름() |
| • 인스턴스의 멤버 변수 참조 | 인스턴스 변수 이름.멤버 변수 이름 |
| • 인스턴스의 메소드 실행 | 인스턴스 변수 이름.메소드 이름() |

3) 클래스 정의 및 인스턴스 기본 사용법

사용 예 1)

```
class Car1:
    model="SM3"
    price=2000

    def sale(self,price):
        self.price=self.price - price

new_car1 = Car1()
new_car1.price = 1500
print(new_car1.price)

new_car1.sale(200)
print(new_car1.model, new_car1.price)
```

클래스 Car1의 멤버 변수 model, price의 값을 지정.

매개변수 price를 갖는 클래스 Car1의 메소드 sale()를 정의.

클래스 Car1을 기반으로 인스턴스 new_car1 생성.

멤버 변수 price를 1500으로 변경.

- new_car1 의 메소드 sale() 를 실행하여 멤버 변수 price 에서 200 을 뺌.

<결과>

1500
SM3 1300

사용 예 2)

```
class Car1_1:
    def __init__(self):
        self.model = "SM5"
        self.price = 3000
    def sale(self, price):
        self.price = self.price - price

new_car1_1 = Car1_1()
new_car1_1.sale(200)

print(new_car1_1.model, new_car1_1.price)
```

- 클래스 Car1 의 초기화(생성자)에서 멤버 변수 model, price 의 값을 지정.
- 매개변수 price 를 갖는 클래스 Car1 의 메소드 sale() 를 정의.
- 클래스 Car1_1 을 기반으로 인스턴스 new_car1_1 생성.
- new_car1_1 의 메소드 sale() 를 실행하여 멤버 변수 price 에서 200 을 뺌.

<결과>

1500
SM3 1300

※ 참고

- 하나의 클래스를 이용하여 여러 개의 객체가 생성될 수 있기 때문에 객체 자신을 가리키도록 self 인자를 사용함.
- 클래스 메소드에서 첫 번째 매개변수는 self 가 되어야 함.
- 인스턴스를 위한 데이터를 저장할 목적이라면 멤버 변수를 생성할 때 멤버 변수 앞에 self. 라는 인자를 적어야 함. self 를 적지 않으면 같은 클래스를 기반으로 하는 인스턴스끼리 멤버 변수에 저장된 데이터를 공유하기 때문에 뜻하지 않는 상황이 발생할 수도 있음.

3. 정보 은닉

1) 기능

- 멤버 변수의 직접적인 접근을 차단.
- 메소드에서 허용하는 방법으로만 멤버 변수의 접근을 허용하여 논리적인 오류를 방지하고, 객체의 안정성을 높임.

2) 사용 예

```
class Car2:
    def __init__(self, model, price):
        self.model = model
        self.price = price

    def sale(self, price):
        self.price = self.price - price

    def get_model(self):
        return self.model

    def get_price(self):
        return self.price

new_car2 = Car2("Sonata", 2500)
new_car2.sale(200)

print(new_car2.get_model(), new_car2.get_price())
```

- Car2 클래스를 기반으로 인스턴스를 생성할 때 매개변수로 전달한 값을 이용하여 멤버 변수 값을 지정.
- 멤버 변수 값을 참조할 때 멤버 변수를 직접 참조하는 것이 아니라 메소드를 이용하여 참조하게 함.

<결과>

Sonata 2300

4. 상속(Inheritance)

1) 의미

- 이미 만들어진 클래스의 멤버 변수와 메소드를 그대로 이어 받아서 필요한 부분만 재정의하거나 추가하여 새로운 클래스를 만드는 것.
- 상속을 해 주는 클래스 : 부모 클래스, 슈퍼 클래스, 상위 클래스라는 용어로 불림.
- 상속을 받는 클래스 : 자식 클래스, 서브 클래스, 하위 클래스라는 용어로 불림.

2) 단일 상속

```
#상속
class Dad:
    def sing(self):
        print("노래를 잘 한다")

class Child(Dad):
    def dance(self):
        print("춤을 잘 춘다")

c = Child()
c.sing()
c.dance()
```



Child 클래스는 Dad 클래스를 상속 받았기 때문에 sing 와 dance 메소드 모두 실행 가능.

<결과>

노래를 잘 한다
춤을 잘 춘다

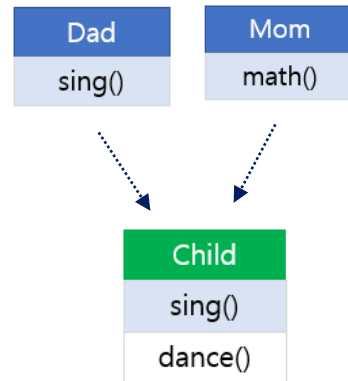
3) 다중 상속 : 부모 클래스가 두 개 이상

```
#다중 상속
class Dad:
    def sing(self):
        print("노래를 잘 한다")

class Mom:
    def math(self):
        print("수학을 잘 한다")

class Child(Dad, Mom):
    def dance(self):
        print("춤을 잘 춘다")

c = Child()
c.sing()
c.math()
c.dance()
```



Child 클래스는 Dad 클래스와 Mom 클래스를 상속 받았기 때문에 sing, math 와 dance 메소드 모두 실행 가능.

<결과>

노래를 잘 한다
수학을 잘 한다
춤을 잘 춘다

5. 메소드 오버라이딩(Overriding)

1) 의미

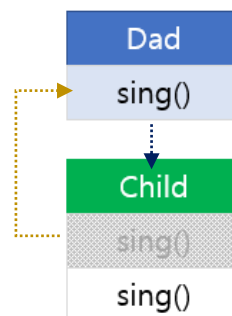
- 자식 클래스에서 부모 클래스에 있는 메소드를 다시 정의하여 부모 클래스에 있는 메소드와 다른 기능을 하도록 하는 것. 일종의 기능 upgrade.
- 만일 자식 클래스에서 메소드 오버라이딩을 했을 때 오버라이딩 한 부모의 메소드를 사용하고 싶다면 자식 클래스에서 super().메소드() 를 호출.

2) 메소드 오버라이딩의 예

```
class Dad:
    def sing(self):
        print("노래를 잘 한다")

class Child(Dad):
    def sing(self):
        super().sing()
        print("랩을 잘 한다.")

c = Child()
c.sing()
```



Child 클래스에서 sing() 메소드를 오버라이딩 함으로써 Dad 클래스의 sing() 메소드를

가리고 Child 클래스에서 재정의한 sing() 메소드의 내용으로 명령이 실행됨.

2

람다, 아이터레이터, 제너레이터, 추상클래스, 뮤터블

1. 람다(lambda)

1) 정의

- 함수 이름에 대한 정의 없이 한 줄로 간단히 표현해서 사용하는 함수.

lambda 매개 변수 : 리턴 값

2) 함수 정의 vs 람다

```
#람다 함수식
def func(a, b):
    return a+b

print(func(10, 20))

s1=lambda a,b:a+b
print(s1(10, 20))
```

- func() 함수를 정의한 후 인수로 10, 20 을 전달하여 호출
- 람다 표현식을 사용하여 함수를 정의한 것을 s1 변수에 할당하고 인수로 10, 20 을 전달하여 호출.

<결과>

30
30

3) 람다를 객체의 메소드로 추가

```
#객체에 람다를 이용해 메소드 추가
class Test:
    pass

t=Test()
t.add_method=lambda:print("메소드 추가")
t.add_method()
```

- 내용이 없는 Test 클래스 생성.
- Test 클래스의 인스턴스 t 를 생성한 후 add_method 라는 메소드를 람다를 이용하여 추가.

2. 스페셜 메소드(Special Method)

1) 정의

- 정해진 특정 시점에 호출되는 메소드.

2) 종류

<code>__init__</code>	객체 생성 시 자동으로 호출되는 메소드
<code>__len__</code>	<code>len()</code> 가 호출되면 자동으로 호출
<code>__iter__</code>	<code>iter()</code> 가 호출되면 자동으로 호출
<code>__str__</code>	<code>str()</code> 이 호출되면 자동으로 호출

3. iterable / iterator

1) iterable 객체의 정의

- 반복 가능한 객체. for 문을 이용하여 객체 안의 항목을 하나씩 가져올 수 있음.
- 대표적인 예 : string, list, tuple, dict, set, range()의 결과값

2) 사용 예

```
#iterable 객체
for i in range(3):
    print(i, end=" ")
```

- range() 함수의 결과 값을 하나씩 i로 받아서 출력.
0 1 2

```
for i in ['토요일', '일요일']:
    print(i, end=" ")
```

- 리스트의 항목을 하나씩 i로 받아서 출력.
토요일 일요일

➔ iterable 객체를 print() 함수를 이용해 출력하면 iterable 객체 전체가 출력되고, iterable 객체 안의 항목을 하나씩 가져오기 위해서는 for 문을 사용해야 함.

3) iterator 객체

iterable 한 객체가 iter() 함수를 통해 iterator 객체로 생성됨.

```
#iterator 객체
x = ['토요일', '일요일']
print(type(x))
x = iter(x)
print(type(x))
y = next(x)
print(y)
y = next(x)
print(y)
y = next(x)
print(y)
```

- 리스트 x의 타입 출력.
<class 'list'>

- 리스트 x를 iterator 객체로 변환하여 타입 출력.
<class 'list_iterator'>

- next() 함수를 이용하여 iterator 객체인 x의 첫 번째 항목을 출력.
토요일

- next() 함수를 이용하여 iterator 객체인 x의 두 번째 항목을 출력.
일요일

- next() 함수를 이용하여 iterator 객체인 x 의 세 번째 항목을 출력. → 항목이 두 개 밖에 없으므로 StopIteration 예외 발생.

```
iterator.py", line 15, in <module>
    y = next(x)
StopIteration
```

→ iterator 객체로 변환하게 되면 필요할 때에 객체 안의 항목을 하나씩 가져와 사용 가능.

4. 제너레이터(Generator)

1) 정의

- iterator 를 생성하는 함수.
- 일반적인 함수와 같은 형태를 갖고 있으나 return 대신에 yield 를 사용.
- 제너레이터 함수 실행 중에 yield 문을 만나면 그 상태로 정지하고 값을 호출한 쪽으로 반환. next() 가 호출될 때마다 제너레이터의 실행이 중지된 곳에서 실행이 재개됨.
- next() 함수를 실행할 때 값을 생성하여 메모리에 적재하므로 효율적으로 메모리를 사용할 수 있음.

2) 제너레이터 사용 예

```
#제너레이터 함수
def gen():
    print("첫 번째")
    yield
    print("두 번째")
    yield 2

g = gen()
y1 = next(g)
print(y1)

y2 = next(g)
print(y2)
```

- 제너레이터 함수의 객체 g 를 생성하여 next() 함수를 이용하여 호출하면 제너레이터 함수의 첫 번째 yield 문까지의 명령이 실행된 후 호출한 곳으로 리턴.
- 두 번째 next() 함수에 의해 제너레이터가 다시 호출되면 실행이 중지된 곳부터 실행이 재개되고 두 번째 yield 문까지의 명령이 실행된 후 호출한 곳으로 리턴.
- 제너레이터가 종료되면 자동으로 'StopIteration' exception 을 발생시킴.

<결과>

```
첫 번째
None
두 번째
2
```

5. 추상 클래스(Abstract Class)

1) 정의

- 메소드의 목록만 가진 클래스(추상 메소드가 하나 이상 있는 클래스)
- 상속 받는 자식 클래스에서 부모의 추상 메소드를 필수적으로 구현해야 함.
- 추상 클래스는 인스턴스를 생성할 수 없음.
- 추상 클래스는 상속을 위해서 사용.
- 자식 클래스에서 추상 클래스의 메소드를 정의하지 않으면 오류 발생.
(`TypeError: Can't instantiate abstract class 추상 클래스 명 with abstract method move`)

2) abc 모듈

- 추상 베이스 클래스(ABC : Abstract Base Class)를 정의하기 위한 기초를 제공하는 모듈.
- 추상 클래스를 사용하려면 반드시 abc 모듈을 import 하고, 생성하는 추상 클래스의 메타 클래스는 ABCMeta 로 지정해야 함..

3) 추상 클래스 작성 예

```
from abc import *
class Animal(metaclass=ABCMeta):
    @abstractmethod
    def move(self):
        pass
class Human(Animal):
    def move(self):
        print("두 발로 움직입니다")
class Dog(Animal):
    def move(self):
        print("네 발로 움직입니다")

h = Human()
h.move()

d = Dog()
d.move()
```

- 추상 클래스 사용을 위해 abc 모듈 import
- 추상 클래스 Animal 을 생성하고, '@abstractmethod' 데코레이터를 사용하여 move() 메소드를 추상 메소드로 지정.
- Animal 클래스를 상속받는 Human 클래스를 생성하고 move() 메소드를 오버라이드.
- Animal 클래스를 상속받는 Dog 클래스를 생성하고 move() 메소드를 오버라이드.

6. mutable / immutable

1) 의미

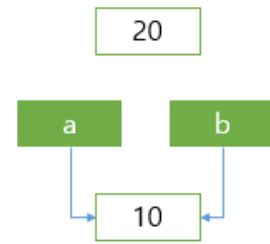
immutable (수정 불가능한 객체)	mutable (수정 가능한 객체)
숫자 형 : int, bool, float 시퀀스 형 : str, tuple	시퀀스 형 : list 집합 형 : set 매핑 형 : dict

2) immutable 예

```
#immutable
a = 10
b = a
print(id(a), a)
print(id(b), b)

a = 20
print(id(a), a)
print(id(b), b)
```

- 10 을 값으로 갖는 정수 객체를 생성하고 a 가 그 객체를 가리키게 함.
- a 가 가리키는 것을 b 가 가리키도록 지정.
- 20 을 값으로 갖는 정수 객체를 생성하고 a 가 그 객체를 가리키도록 변경.



※ id() : 객체가 차지하는 메모리 주소를 return 하는 함수.

<결과>

```
2496443345488 10
2496443345488 10
2496443345808 20
2496443345488 10
```

- a 의 값을 20 으로 변경하면 a 가 가리키는 주소가 바뀜.

3) mutable 예

```
#mutable
c = [1,2,3]
d = c
print(id(c), c)
print(id(d), d)

c[2] = 4
print(id(c), c)
print(id(d), d)
```

- 리스트 [1,2,3]을 생성하여 c 가 그 객체를 가리키게 함.
- c 가 가리키는 것을 d 가 가리키도록 지정.
- c 가 가리키는 리스트의 세 번째 값을 4 로 변경.

c		
1	2	3

<결과>

```
2349255939328 [1, 2, 3]
2349255939328 [1, 2, 3]
2349255939328 [1, 2, 4]
2349255939328 [1, 2, 4]
```

- 리스트의 세 번째 값을 변경해도 두 변수가 가리키는 주소는 동일.