

파이썬 입문

한국폴리텍대학

2023.05

다중 상속 (multiple Inheritance) – 생성자(constructor)

```
class p1:  
    def __init__(self):  
        self.a = 100
```

```
class p2:  
    def __init__(self):  
        self.b = 200
```

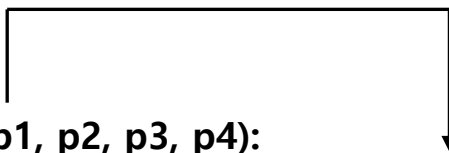
```
class p3:  
    def __init__(self):  
        self.c = 300
```

```
class p4:  
    def __init__(self):  
        self.d = 400
```

```
class p5(p1, p2, p3, p4):  
    def __init__(self):  
        p1.__init__(self)  
        p2.__init__(self)  
        p3.__init__(self)  
        p4.__init__(self)
```

```
pp = p5()  
print(pp.a, pp.b, pp.c, pp.d)    # 100, 200, 300, 400
```

다중 상속에서 2개이상의 부모클래스
__init__()을 실행시켜야 하는 경우,
super().__init__() 대신에 (1개만 가능)
부모클래스명1.__init__(self),
부모클래스명2.__init__(self),
..... 로 사용한다



```
class p5(p1, p2, p3, p4):  
    def __init__(self):  
        super().__init__() : 맨 앞 부모클래스 __init__만 실행  
        p2.__init__(self)  
        p3.__init__(self)  
        p4.__init__(self)
```

다중 상속 (multiple Inheritance) – 생성자(constructor)

```
class p1:  
    def __init__(self):  
        self.a = 100
```

```
class p2:  
    def __init__(self):  
        self.b = 200
```

```
class p3(p1, p2):  
    pass
```

```
pp = p3()
```

```
print(pp.a)          # 100  
#print(pp.a, pp.b)  # pp.b 에러
```

1. 자식 클래스에 생성자가 없는 경우,
 맨 앞 부모클래스의 생성자만 실행 (p1)
 --> pp.b 에러 (선언 안됨)

```
class p1:  
    def __init__(self):  
        self.a = 100
```

```
class p2:  
    def __init__(self):  
        self.b = 200
```

```
class p3(p1, p2):  
    def __init__(self):  
        pass
```

```
pp = p3()
```

```
print(pp.a, pp.b)    # pp.a, pp.b 에러
```

1. 자식 클래스에 생성자가 있는 경우
 부모 클래스의 생성자는 실행안됨
 --> pp.a, pp.b 모두 에러 (선언 안됨)

다중 상속 (multiple Inheritance) - 오버라이딩 (overriding)

```
class p1:  
    def __init__(self):  
        self.a = 100
```

```
class p2:  
    def __init__(self):  
        self.b = 200
```

```
class p3(p1, p2):  
    def __init__(self):  
        self.a = 500
```

--> __init__ 오버라이딩

```
pp = p3()
```

```
print(pp.a)          # 500  
#print(pp.a, pp.b)  # pp.b 에러
```

overriding : 최우선시 되는, 재정의
부모 클래스의 메소드를 재정의하여 사용

super() 함수 : 부모 클래스의 메소드를 사용

1. 생성자의 경우 맨 앞 부모 클래스만 가능
2. 동일한 메소드가 존재하는 경우 앞에 있는 부모 클래스의 메소드 사용

```
super().move()  
p3.move(self) --> 부모 클래스를 지정하면 됨
```

다중 상속 (multiple Inheritance) – super() 함수

```
class p1:
    def __init__(self):
        self.a = 100

    def move1(self):
        print("move1")
```

```
class p2:
    def __init__(self):
        self.b = 200

    def move2(self):
        print("move2")
```

```
class p3(p1, p2):
    def __init__(self):          --> __init__() 오버라이딩
        self.a = 500

    def move1(self):            --> move1() 오버라이딩
        print("move100")
```

```
pp = p3()
```

```
pp.move1()      # move100
```

```
class p1:
    def __init__(self):
        self.a = 100

    def move1(self):
        print("move1")
```

```
class p2:
    def __init__(self):
        self.b = 200

    def move2(self):
        print("move2")
```

```
class p3(p1, p2):
    def __init__(self):          --> __init__() 오버라이딩
        self.a = 500

    def move1(self):            --> move1() 오버라이딩
        super().move2()         --> self 없음
        p2.move2(self)          --> self 있음
```

```
pp = p3()
```

```
pp.move1()      # move2 W move2
```

:= 연산자

```
a = 'hello world'
```

```
b = len(a)
```

```
if b > 3:
```

```
    print('ok')
```

```
a = 'hello world'
```

```
if b := len(a) > 3:
```

```
    print('ok')
```

walrus operator (바다코끼리 연산자)

--> python v3.8 이상부터 가능

함수 매개변수 순서

```
def func ( a, b, c ):  
    a1 = a; b1 = b; c1 = c  
    print(a1, b1, c1)  
func(10, 20, 30)      # 10, 20, 30
```

```
def func ( a, b, c='100' ):  
    a1 = a; b1 = b; c1 = c  
    print(a1, b1, c1)  
func(10, 20, 30)      # 10, 20, 30  
func(10, 20)           # 10, 20, 100  
func(10)               # error
```

```
def func ( a, b='100', c ):  
    a1 = a; b1 = b; c1 = c  
    print(a1, b1, c1)  
func(10, 20, 30)      # error  
                        # non-default argument follows default argument  
                        # default argument: 기본값이 있는 인수, non-default argument: 기본값이 없는 인수  
func(10, 20, c=30)    # error
```

매개변수 순서

1. 위치 인수 (non-default(or positional) arguments) : a
2. 기본값 인수 (default arguments) : a = 100
3. 가변 인수 (variable length positional arguments) : *args
4. 키워드 인수 (keyword arguments) : kwarg
5. 키워드 가변 인수 (variable length keyword arguments) : **kwargs

함수 매개변수 순서

```
def func ( a, b='100', *c ):
    print(a, b, c[0], c[1], c[2])
func(10, 20, 30, 40, 50)      # 10, 20, 30, 40, 50
```

```
def func ( a, b='100', *c ):
    print(a, b, c)
func(10)                      # 10, 100, ()
```

```
def func ( a, b='100', *c, d, **e ):
    print(a, b, c, d, e)
func(10, 20, 30, 40, 50, d=60, aa=1, bb=2, cc=3 )  # 10, 20, (30, 40, 50), 60, {'aa':1, 'bb':2, 'cc':3}
```

키워드 인수

```
data= {'aa':1, 'bb':2, 'cc':3}
func(10, 20, 30, 40, 50, d=60, **data)      # 10, 20, (30, 40, 50), 60, {'aa':1, 'bb':2, 'cc':3}
```

딕셔너리 형태로도 가능 {'aa':1, 'bb':2, 'cc':3}

```
def func ( a, b='100', *c, d ):
    pass
```

```
func(10, 20, 30, 40)          # error
func(10, 20, 30, d=40)        # 10, 20, 30, 40
```


데코레이터 (@decorator)

```
def decorator1(func):  
    def wrapper():  
        print(func.__name__, 'start')  
        func()    # hello()  
        print(func.__name__, 'end')  
    return wrapper  
  
@decorator1  
def hello():  
    print('hello')  
  
hello()           # hello start  
                  # hello  
                  # hello end
```

데코레이터 - @decorator

```
def decorator1(func):  
    def wrapper():  
        print(func.__name__, 'start1')  
        func()  
        print(func.__name, 'end1')  
    return wrapper
```

```
def decorator2(func):  
    def wrapper2():  
        print(func.__name, 'start2')  
        func()  
        print(func.__name, 'end2')  
    return wrapper2
```

```
@decorator1  
@decorator2  
def hello():  
    print('hello')
```

```
hello()           # wrapper2 start1  
                  # hello start2  
                  # hello  
                  # hello end2  
                  # wrapper2 end1
```

```
def a(func):  
    def wrapper():  
        print(func.__name__, 'start')  
        func()  
        print(func.__name__, 'end')  
    return wrapper
```

```
def test():  
    print('test')
```

```
b = a(test)  
b()
```

데코레이터 - @Property

```
class Person:
    def __init__(self):
        self.__age = 0

    def get_age(self):          # getter
        return self.__age

    def set_age(self, value):   # setter
        self.__age = value

kim = Person()
kim.set_age(20)
print(kim.get_age())          # 20
```

```
class Person:
    def __init__(self):
        self.__age = 0

    @property
    def age(self):              # getter
        return self.__age

    @age.setter
    def age(self, value):       # setter
        self.__age = value

kim = Person()
kim.age = 20
print(kim.age)                 # 20
```

getter에는 @property 데코레이터를,
Setter에는 @age.setter 데코레이터를 사용

값을 저장할때는 kim.age = 20 (kim.set_age(20) 아님)
값을 가져올때는 kim.age (kim.get_age() 아님)

데코레이터 - @staticmethod, @classmethod

```
class Person:

    def a(self):
        pass

    #def a1():
    #    pass

    @staticmethod
    def b():                --> self 없음
        pass

print(type(Person.a), type(Person.b)) # function, function

c = Person()
print(type(c.a), type(c.b))           # method, function
#print(type(c.a1))                     # method
#print(c.a1())                         # error
#print(Person.a1())

# staticmethod는 class와 상관없는 독립적인 함수
# 클래스, 인스턴스에서 모두 접근 가능함
```

```
class Person:
    default = 'father'
    def __init__(self):
        self.data = self.default
        print(self.data)

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

    @staticmethod
    def static_person():
        return Person()

class Person2(Person):
    default = 'mother'

Person2.class_person() # mother
Person2.static_person() # father

# 인스턴스 메소드 func(self) --> 인스턴스 변수에 접근 가능
# 클래스 메소드 func(cls)   @classmethod 데코레이터 사용
                             --> 클래스 변수에 접근 가능
# 정적 메소드 func()       @staticmethod 데코레이터 사용
                             --> 클래스 변수에 접근 불가능
```

주사위 게임

(1) 50명의 플레이어가 참가한다

(2) 각 플레이어는 10개의 코인을 가진다.

게임 1회당 코인 1개가 소진된다. 코인이 없는 경우 게임을 1회 쉬고, 랜덤으로 1개에서 10개의 코인이 충전된다.

(2) 각 플레이어가 2개의 주사위를 던진다.

만약 2개의 주사위 수가 같은 경우, 한 번 더 던지고, 수를 모두 더한다

(예) 주사위 수 (2, 6) = 8

(예) 주사위 수 (3, 3) + 주사위 수 (4, 3) = 13

(3) 가장 높은 수가 나온 플레이어들은 주사위 수 x 2배의 점수를 얻고, 가장 낮은 수가 나온 플레이어들은 30점을 잃는다.

나머지 플레이어들은 주사위 수만큼 점수를 얻는다.

(4) 점수 상황을 화면에 박스 그래프로 표시해준다. 박스 최소값은 0, 최대값은 1000)

(5) 1000점을 가장 먼저 얻는 플레이어가 승리한다.

(6) 게임은 0.1초마다 반복된다

미니 프로젝트

```
import numpy as np
import cv2 as cv
import random as rd

width = 1024
height = 600
depth = 3

thickness = -1 # e.g. -1, 1, 10, .....
offset_x = 50
offset_y = 20
x = 55
y = 22
screen_start_x = 100
screen_start_y = 500
bgr = (0, 255, 0)
count = 15

while True:
    img = np.zeros(shape = (height, width, depth)) # img = np.zeros((height, width, depth), np.uint8)

    for i in range(count):
        bar_cnt = rd.randint(1,6)

        for j in range(bar_cnt):
            start_x = screen_start_x + i * x
            start_y = screen_start_y - j * y
            cv.rectangle(img, (start_x, start_y), (start_x + offset_x, start_y + offset_y), bgr, thickness)

    cv.imshow("test",img)
    cv.waitKey(100)
```