Unit 42.

Status	완료
담당자	
마감일	
완료일	@2022년 12월 2일

Unit 42. 데코레이터 사용하기

@로 시작하는 것들이 데코레이터

```
class Calc:
    @staticmethod    # 데코레이터
    def add(a, b):
        print(a + b)
```

42.1 데코레이터 만들기

데코레이터는 함수를 수정하지 않은 상태에서 추가 기능을 구현할 때 사용

예를 들어서 함수의 시작과 끝을 출력하고 싶다면 다음과 같이 함수 시작, 끝 부분에 print를 넣어야 하는데 함수가 많아진다면 매우 번거로운데 이러 부분을 해결하기 위해 데코레이터 사용

```
def trace(func):                              # 호출할 함수를 매개변수로 받음
    def wrapper():                            # 호출할 함수를 감싸는 함수
        print(func.__name__, '함수 시작')    # __name__으로 함수 이름 출력
        func()                                # 매개변수로 받은 함수를 호출
        print(func.__name__, '함수 끝')
    return wrapper                            # wrapper 함수 반환

def hello():
    print('hello')

def world():
    print('world')

trace_hello = trace(hello)    # 데코레이터에 호출할 함수를 넣음
trace_hello()                 # 반환된 함수를 호출
trace_world = trace(world)    # 데코레이터에 호출할 함수를 넣음
trace_world()                 # 반환된 함수를 호출
```

42.1.1 @로 데코레이터 사용하기

```
@데코레이터
def 함수이름():
    코드
```

```
def trace(func):                              # 호출할 함수를 매개변수로 받음
    def wrapper():
        print(func.__name__, '함수 시작')    # __name__으로 함수 이름 출력
        func()                                # 매개변수로 받은 함수를 호출
        print(func.__name__, '함수 끝')
    return wrapper                            # wrapper 함수 반환

@trace    # @데코레이터
def hello():
    print('hello')

@trace    # @데코레이터
def world():
    print('world')

hello()    # 함수를 그대로 호출
world()    # 함수를 그대로 호출
```

hello와 world 함수 위에 @trace를 붙인 뒤에 hello와 world 함수를 그대로 호출

데코레이터는 함수를 감싸는 형태로 구성되어 있다.

42.2 매개변수와 반환값을 처리하는 데코레이터 만들기

함수의 매개변수와 반환값을 출력하는 데코레이터이다.

```
def trace(func):          # 호출할 함수를 매개변수로 받음
    def wrapper(a, b):    # 호출할 함수 add(a, b)의 매개변수와 똑같이 지정
        r = func(a, b)    # func에 매개변수 a, b를 넣어서 호출하고 반환값을 변수에 저장
        print('{0}(a={1}, b={2}) -> {3}'.format(func.__name__, a, b, r))  # 매개변수와 반환값 출력
        return r          # func의 반환값을 반환
    return wrapper        # wrapper 함수 반환

@trace              # @데코레이터
def add(a, b):      # 매개변수는 두 개
    return a + b    # 매개변수 두 개를 더해서 반환

print(add(10, 20))
```

데코레이터를 만들 때는 먼저 안쪽 wrapper 함수의 매개변수를 호출할 함수 add(a, b)의 매개변수와 똑같이 만들어준다.

```
def trace(func):          # 호출할 함수를 매개변수로 받음
    def wrapper(a, b):    # 호출할 함수 add(a, b)의 매개변수와 똑같이 지정
```

wrapper 함수 안에서는 func를 호출하고 반환값을 변수에 저장하고 print로 매개변수와 반환값을 출력한다.

add 함수는 두 수를 더해서 반환해야 하므로 func의 반환값을 return으로 반환해준다.

```
def trace(func):          # 호출할 함수를 매개변수로 받음
    def wrapper(a, b):    # 호출할 함수 add(a, b)의 매개변수와 똑같이 지정
        r = func(a, b)    # func에 매개변수 a, b를 넣어서 호출하고 반환값을 변수에 저장
        print('{0}(a={1}, b={2}) -> {3}'.format(func.__name__, a, b, r))    # 매개변수와 반환값 출력
        return r          # func의 반환값을 반환
    return wrapper        # wrapper 함수 반환
```

데코레이터를 사용할 때는 @로 함수 위에 지정해주면 됩니다. 또한, @로 데코레이터를 사용했으므로 add 함수는 그대로 호출해준다.

```
@trace              # @데코레이터
def add(a, b):      # 매개변수는 두 개
    return a + b    # 매개변수 두 개를 더해서 반환
```

42.2.1 가변 인수 함수 데코레이터

매개변수(인수)가 고정되지 않은 함수는 처리하는 방법은

여기서는 wrapper 함수를 가변 인수 함수로 만들면 됩니다.

get_max 함수와 get_min 함수는 가변 인수 함수이기 때문에 데코레이터도 가변 인수 함수로 만들어준다.

```
def trace(func):                    # 호출할 함수를 매개변수로 받음
    def wrapper(*args, **kwargs):    # 가변 인수 함수로 만듦
        r = func(*args, **kwargs)    # func에 args, kwargs를 언패킹하여 넣어줌
        print('{0}(args={1}, kwargs={2}) -> {3}'.format(func.__name__, args, kwargs, r))
                                     # 매개변수와 반환값 출력
        return r                     # func의 반환값을 반환
    return wrapper                   # wrapper 함수 반환

@trace                  # @데코레이터
def get_max(*args):     # 위치 인수를 사용하는 가변 인수 함수
    return max(args)

@trace                  # @데코레이터
def get_min(**kwargs):  # 키워드 인수를 사용하는 가변 인수 함수
    return min(kwargs.values())

print(get_max(10, 20))
print(get_min(x=10, y=20, z=30))
```

wrapper 함수 안에서는 func를 호출해주는데 args는 튜플이고, kwargs는 딕셔너리이므로 func에 넣을 때는 언패킹하여 넣어준다.

```
def trace(func):                    # 호출할 함수를 매개변수로 받음
    def wrapper(*args, **kwargs):    # 가변 인수 함수로 만듦
        r = func(*args, **kwargs)    # func에 args, kwargs를 언패킹하여 넣어줌
        print('{0}(args={1}, kwargs={2}) -> {3}'.format(func.__name__, args, kwargs, r))
                                     # 매개변수와 반환값 출력
        return r                     # func의 반환값을 반환
    return wrapper                   # wrapper 함수 반환
```

데코레이터 trace는 위치 인수와 키워드 인수를 모두 처리할 수 있습니다. 따라서 가변 인수 함수뿐만 아니라 일반적인 함수에도 사용할 수 있다.

```
>>> @trace
... def add(a, b):
...     return a + b
...
>>> add(10, 20)
add(args=(10, 20), kwargs={}) -> 30
30
```

42.3 매개변수가 있는 데코레이터 만들기

매개변수가 있는 데코레이터는 값을 지정해서 동작을 바꿀 수 있습니다.

```
def is_multiple(x):              # 데코레이터가 사용할 매개변수를 지정
    def real_decorator(func):    # 호출할 함수를 매개변수로 받음
        def wrapper(a, b):       # 호출할 함수의 매개변수와 똑같이 지정
            r = func(a, b)       # func를 호출하고 반환값을 변수에 저장
            if r % x == 0:       # func의 반환값이 x의 배수인지 확인
                print('{0}의 반환값은 {1}의 배수입니다.'.format(func.__name__, x))
            else:
                print('{0}의 반환값은 {1}의 배수가 아닙니다.'.format(func.__name__, x))
            return r             # func의 반환값을 반환
        return wrapper           # wrapper 함수 반환
    return real_decorator        # real_decorator 함수 반환

@is_multiple(3)     # @데코레이터(인수)
def add(a, b):
    return a + b

print(add(10, 20))
print(add(2, 5))
```

매개변수가 있는 데코레이터를 만들 때는 함수를 하나 더 만들어야 한다.

함수를 두 개 만들었으므로 함수를 만든 뒤에 return으로 두 함수를 반환해준다.

데코레이터를 사용할 때는 데코레이터에 ( )(괄호)를 붙인 뒤 인수를 넣어주면 된다.

```
@데코레이터(인수)
def 함수이름():
    코드
```

```
@is_multiple(3)     # @데코레이터(인수)
def add(a, b):
    return a + b
```

42.4 클래스로 데코레이터 만들기

클래스로도 데코레이터 만들 수 있다.

클래스를 활용할 때는 인스턴스를 함수처럼 호출하게 해주는 **call** 메서드를 구현해야 한다.

클래스로 데코레이터를 만들 때는 먼저 **init** 메서드를 만들고 호출할 함수를 초깃값으로 받는다.

매개변수로 받은 함수를 속성으로 저장

```
class Trace:
    def __init__(self, func):    # 호출할 함수를 인스턴스의 초깃값으로 받음
        self.func = func         # 호출할 함수를 속성 func에 저장

    def __call__(self):
        print(self.func.__name__, '함수 시작')    # __name__으로 함수 이름 출력
```

```
        self.func()                             # 속성 func에 저장된 함수를 호출
        print(self.func.__name__, '함수 끝')

@Trace    # @데코레이터
def hello():
    print('hello')

hello()    # 함수를 그대로 호출
```

```
class Trace:
    def __init__(self, func):    # 호출할 함수를 인스턴스의 초깃값으로 받음
        self.func = func         # 호출할 함수를 속성 func에 저장
```

테코레이터를 사용할때는 호출할 함수 위에 @을 붙이고 데코레이터를 지정하면 된다.

```
@데코레이터
def 함수이름():
    코드
```

```
@Trace    # @데코레이터
def hello():
    print('hello')
```

클래스로 만든 데코레이터는 @을 지정하지 않고, 데코레이터의 반환값을 호출하는 방식으로도 사용할 수 있다.

```
def hello():    # @데코레이터를 지정하지 않음
    print('hello')

trace_hello = Trace(hello)    # 데코레이터에 호출할 함수를 넣어서 인스턴스 생성
trace_hello()                 # 인스턴스를 호출. __call__ 메서드가 호출됨
```

42.5 클래스로 매개변수와 반환값을 처리하는 데코레이터 만들기

클래스로 만든 데코레이터도 매개변수와 반환값을 처리할 수 있다.

```
class Trace:
    def __init__(self, func):    # 호출할 함수를 인스턴스의 초깃값으로 받음
        self.func = func         # 호출할 함수를 속성 func에 저장

    def __call__(self, *args, **kwargs):    # 호출할 함수의 매개변수를 처리
        r = self.func(*args, **kwargs) # self.func에 매개변수를 넣어서 호출하고 반환값을 변수에 저장
        print('{0}(args={1}, kwargs={2}) -> {3}'.format(self.func.__name__, args, kwargs, r))
                                       # 매개변수와 반환값 출력
        return r                       # self.func의 반환값을 반환

@Trace    # @데코레이터
def add(a, b):
    return a + b

print(add(10, 20))
print(add(a=10, b=20))
```

클래스로 매개변수와 반환값을 처리하는 데코레이터를 만들 때는 **call** 메서드에 매개변수를 지정하고, self.func에 매개변수를 넣어서 호출한 뒤에 반환값을 반환해주면 된다.

```
    def __call__(self, *args, **kwargs):    # 호출할 함수의 매개변수를 처리
        r = self.func(*args, **kwargs) # self.func에 매개변수를 넣어서 호출하고 반환값을 변수에 저장
        print('{0}(args={1}, kwargs={2}) -> {3}'.format(self.func.__name__, args, kwargs, r))
                                       # 매개변수와 반환값 출력
        return r                       # self.func의 반환값을 반환
```

고정된 매개변수를 사용할 때는 def **call**(self, a, b):처럼 만들어도 된다.

42.5.1 클래스로 매개변수가 있는 데코레이터 만들기

함수의 반환값이 특정 수의 배수인지 확인하는 데코레이터

```
class IsMultiple:
    def __init__(self, x):          # 데코레이터가 사용할 매개변수를 초깃값으로 받음
        self.x = x                  # 매개변수를 속성 x에 저장

    def __call__(self, func):       # 호출할 함수를 매개변수로 받음
        def wrapper(a, b):          # 호출할 함수의 매개변수와 똑같이 지정(가변 인수로 작성해도 됨)
            r = func(a, b)          # func를 호출하고 반환값을 변수에 저장
            if r % self.x == 0:     # func의 반환값이 self.x의 배수인지 확인
                print('{0}의 반환값은 {1}의 배수입니다.'.format(func.__name__, self.x))
            else:
                print('{0}의 반환값은 {1}의 배수가 아닙니다.'.format(func.__name__, self.x))
            return r                # func의 반환값을 반환
        return wrapper              # wrapper 함수 반환

@IsMultiple(3)    # 데코레이터(인수)
def add(a, b):
    return a + b

print(add(10, 20))
print(add(2, 5))
```

init 메서드에서 데코레이터가 사용할 매개변수를 초깃값으로 받고 매개변수를 _ call _ 메서드에서 사용할 수 있도록 속성에 저장

```
 def __init__(self, x):         # 데코레이터가 사용할 매개변수를 초깃값으로 받음
        self.x = x                  # 매개변수를 속성 x에 저장
```

지금까지 __init__에서 호출할 함수를 매개변수로 받았는데 여기서는 데코레이터가 사용할 매개변수를 받는다.

```
 def __call__(self, func):      # 호출할 함수를 매개변수로 받음
        def wrapper(a, b):          # 호출할 함수의 매개변수와 똑같이 지정(가변 인수로 작성해도 됨)
```

데코레이터를 사용할 때는 데코레이터에 ( )(괄호)를 붙인 뒤 인수를 넣어주면 된다.

```
@데코레이터(인수)
def 함수이름():
    코드
```