UNIT 42

데코레이터 사용하기

42 데코레이터 사용하기

데코레이터 사용하기

- 데코레이터는 장식하다, 꾸미다라는 뜻의 decorate에 er(or)을 붙인 말인데 장식하는 도구 정도로 설명할 수 있음
- 클래스에서 메서드를 만들 때 @staticmethod, @classmethod, @abstractmethod 등을 붙였는데, 이렇게 @로 시작하는 것들이 데코레이터임
- 함수(메서드)를 장식한다고 해서 이런 이름이 붙었음

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

42.1 데코레이터 만들기

데코레이터 사용하기

- 데코레이터는 함수를 수정하지 않은 상태에서 추가 기능을 구현할 때 사용함

function_begin_end.py

```
def hello():
    print('hello 함수 시작')
    print('hello')
    print('hello 함수 끝')

def world():
    print('world 함수 시작')
    print('world')
    print('world 함수 끝')

hello()
world()
```

실행 결과

```
hello 함수 시작
hello
hello 함수 끝
world 함수 시작
world
world 함수 끝
```

42.1 데코레이터 만들기

데코레이터 사용하기

- 다음은 함수의 시작과 끝을 출력하는 데코레이터임

decorator_closure.py

```
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
```

실행 결과

```
hello 함수 시작
hello
hello 함수 끝
world 함수 시작
world
world 함수 끝
```

42.1 데코레이터 만들기

데코레이터 사용하기

```
def trace(func):                                # 호출할 함수를 매개변수로 받음
```

```
    def wrapper():                              # 호출할 함수를 감싸는 함수
```

```
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
```

- 함수 안에서 함수를 만들고 반환하는 클로저임

```
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
```

- trace에 다른 함수를 넣은 뒤 반환된 함수를 호출하면 해당 함수의 시작과 끝을 출력할 수 있음

42.1 데코레이터 만들기

데코레이터 사용하기

- 다음과 같이 호출할 함수 위에 @데코레이터 형식으로 지정함

```
@데코레이터
def 함수이름():
    코드
```

42.1 데코레이터 만들기

데코레이터 사용하기

decorator_closure_at_sign.py

```
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
```

실행 결과

```
hello 함수 시작
hello
hello 함수 끝
world 함수 시작
world
world 함수 끝
```

42.1 데코레이터 만들기

@로 데코레이터 사용하기

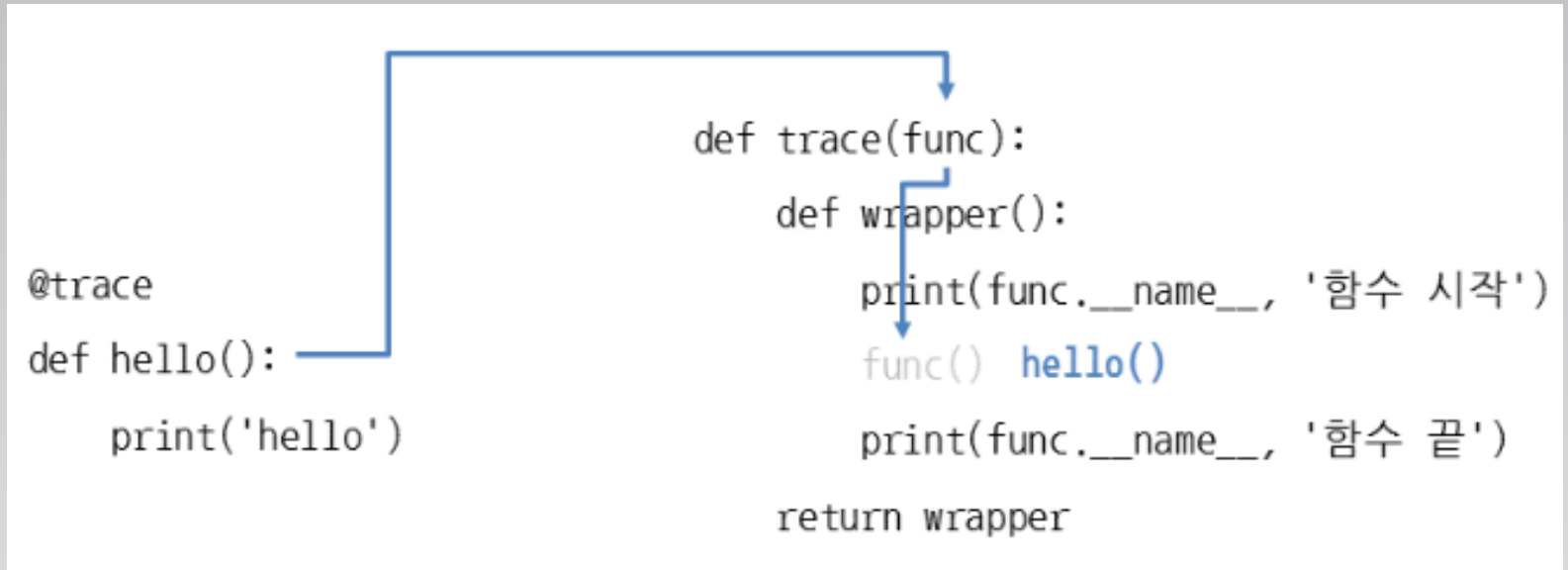
```
@trace    # @데코레이터
def hello():
    print('hello')

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

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

42.1 데코레이터 만들기

▼ 그림 데코레이터



42.2 매개변수와 반환값을 처리하는 데코레이터 만들기

매개변수와 반환값을 처리하는 데코레이터 만들기

- 다음은 함수의 매개변수와 반환값을 출력하는 데코레이터임

decorator_param_return.py

```
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))
```

실행 결과

```
add(a=10, b=20) -> 30
30
```

42.2 매개변수와 반환값을 처리하는 데코레이터 만들기

» 매개변수와 반환값을 처리하는 데코레이터 만들기

- 매개변수와 반환값을 처리하는 데코레이터를 만들 때는 먼저 안쪽 wrapper 함수의 매개변수를 호출할 함수 add(a, b)의 매개변수와 똑같이 만들어줌

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

```
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
```

- wrapper 함수에서 func의 반환값을 반환하지 않으면 add 함수를 호출해도 반환값이 나오지 않으므로 주의해야 함
- wrapper 함수에서 func의 반환값을 출력할 필요가 없으면 return func(a, b)처럼 func를 호출하면서 바로 반환해도 됨

42.2 매개변수와 반환값을 처리하는 데코레이터 만들기

» 매개변수와 반환값을 처리하는 데코레이터 만들기

- 데코레이터를 사용할 때는 @로 함수 위에 지정해주면 됨
- @로 데코레이터를 사용했으므로 add 함수는 그대로 호출해줌

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

42.2 매개변수와 반환값을 처리하는 데코레이터 만들기

가변 인수 함수 데코레이터

- 매개변수(인수)가 고정되지 않은 함수인데 이때는 wrapper 함수를 가변 인수 함수로 만들면됨

decorator_variable_argument.py

```
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))
```

실행 결과

```
get_max(args=(10, 20), kwargs={}) -> 20
20
get_min(args=(), kwargs={'x': 10, 'y': 20, 'z': 30}) -> 10
10
```

42.2 매개변수와 반환값을 처리하는 데코레이터 만들기

가변 인수 함수 데코레이터

- get_max 함수와 get_min 함수는 가변 인수 함수임
- 데코레이터도 가변 인수 함수로 만들어줌
- 위치 인수와 키워드 인수를 모두 받을 수 있도록 *args와 **kwargs를 지정해줌

```
def trace(func):                        # 호출할 함수를 매개변수로 받음
    def wrapper(*args, **kwargs):       # 가변 인수 함수로 만듦
```

```
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
```

42.2 매개변수와 반환값을 처리하는 데코레이터 만들기

가변 인수 함수 데코레이터

- 가변 인수 함수뿐만 아니라 일반적인 함수에도 사용할 수 있음

```
>>> @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 데코레이터 만들기

- 다음은 함수의 반환값이 특정 수의 배수인지 확인하는 데코레이터임

decorator_parameter.py

```
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))
```

실행 결과

```
add의 반환값은 3의 배수입니다.
30
add의 반환값은 3의 배수가 아닙니다.
7
```

42.3 매개변수가 있는 데코레이터 만들기

» 매개변수가 있는 데코레이터 만들기

- 매개변수가 있는 데코레이터를 만들 때는 함수를 하나 더 만들어야 함

```
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
```

```
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
```

42.3 매개변수가 있는 데코레이터 만들기

» 매개변수가 있는 데코레이터 만들기

- real_decorator, wrapper 함수를 두 개 만들었으므로 함수를 만든 뒤에 return으로 두 함수를 반환해줌

```
        return wrapper            # wrapper 함수 반환
    return real_decorator         # real_decorator 함수 반환
```

```
@데코레이터(인수)
def 함수이름():
    코드
```

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

- is_multiple에 다른 숫자를 넣으면 함수의 반환값이 해당 숫자의 배수인지 확인해줌

42.4 클래스로 데코레이터 만들기

클래스로 데코레이터 만들기

- 클래스를 활용할 때는 인스턴스를 함수처럼 호출하게 해주는 __call__ 메서드를 구현해야 함
- 다음은 함수의 시작과 끝을 출력하는 데코레이터임

decorator_class.py

```
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
```

실행 결과

```
hello 함수 시작
hello
hello 함수 끝
```

42.4 클래스로 데코레이터 만들기

클래스로 데코레이터 만들기

```
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')
```

42.4 클래스로 데코레이터 만들기

클래스로 데코레이터 만들기

- 데코레이터를 사용하는 방법은 클로저 형태의 데코레이터와 같음
- 호출할 함수 위에 @을 붙이고 데코레이터를 지정하면 됨

```
@데코레이터
def 함수이름():
    코드
```

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

```
hello()    # 함수를 그대로 호출
```

```
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 매개변수와 반환값을 처리하는 데코레이터 만들기

- 다음은 함수의 매개변수를 출력하는 데코레이터임(여기서는 위치 인수와 키워드 인수를 모두 처리하는 가변 인수로 만들었음)

decorator_class_param_return.py

```
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))
```

실행 결과

```
add(args=(10, 20), kwargs={}) -> 30
30
add(args=(), kwargs={'a': 10, 'b': 20}) -> 30
30
```

42.5 클래스로 매개변수와 반환값을 처리하는 데코레이터 만들기

» 클래스로 매개변수와 반환값을 처리하는 데코레이터 만들기

- 클래스로 매개변수와 반환값을 처리하는 데코레이터를 만들 때는 __call__ 메서드에 매개변수를 지정하고, self.func에 매개변수를 넣어서 호출한 뒤에 반환값을 반환해주면 됨
- 매개변수를 *args, **kwargs로 지정했으므로 self.func에 넣을 때는 언패킹하여 넣어줌

```
    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
```

- 가변 인수를 사용하지 않고, 고정된 매개변수를 사용할 때는 def __call__(self, a, b):처럼 만들어도 됨

42.5 클래스로 매개변수와 반환값을 처리하는 데코레이터 만들기

클래스로 매개변수가 있는 데코레이터 만들기

- 다음은 함수의 반환값이 특정 수의 배수인지 확인하는 데코레이터임

decorator_class_parameter.py

```
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))
```

실행 결과

```
add의 반환값은 3의 배수입니다.
30
add의 반환값은 3의 배수가 아닙니다.
7
```

42.5 클래스로 매개변수와 반환값을 처리하는 데코레이터 만들기

클래스로 매개변수가 있는 데코레이터 만들기

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

- __init__에서 호출할 함수를 매개변수로 받았는데 여기서는 데코레이터가 사용할 매개변수를 받는다는 점 꼭 기억해두자

42.5 클래스로 매개변수와 반환값을 처리하는 데코레이터 만들기

클래스로 매개변수가 있는 데코레이터 만들기

- 이제 __call__ 메서드에서는 호출할 함수를 매개변수로 받음
- __call__ 메서드 안에서 wrapper 함수를 만들어줌
- wrapper 함수의 매개변수는 호출할 함수의 매개변수와 똑같이 지정해줌(가변 인수로 작성해도 됨)

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

42.5 클래스로 매개변수와 반환값을 처리하는 데코레이터 만들기

클래스로 매개변수가 있는 데코레이터 만들기

- 이제 __call__ 메서드에서는 호출할 함수를 매개변수로 받음
- __call__ 메서드 안에서 wrapper 함수를 만들어줌
- wrapper 함수의 매개변수는 호출할 함수의 매개변수와 똑같이 지정해줌(가변 인수로 작성해도 됨)

```
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
```

42.5 클래스로 매개변수와 반환값을 처리하는 데코레이터 만들기

클래스로 매개변수가 있는 데코레이터 만들기

- 데코레이터를 사용할 때는 데코레이터에 ( )(괄호)를 붙인 뒤 인수를 넣어주면 됨

```
@데코레이터(인수)
def 함수이름():
    코드
```

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