Python 3

> @

Head First!

```
def my_decorator(func):
    def wrapper():
        print("Something is happening before the function is called.")
        func()
        print("Something is happening after the function is called.")
    return wrapper


@my_decorator
def say_whee():
    print("Whee!")


say_whee()
```

```
Something is happening before the function is called.
Whee!
Something is happening after the function is called.
```

Decoradores

Un **decorador** es un nombre usado para un patrón de diseño.

Un **decorador** altera dinámicamente el comportamiento de una función, método o clase sin modificar su código fuente o utilizar subclases.

En el mundo de Python la palabra **decorador** se utiliza para notar una conveniencia sintáctica que facilita la aplicación de estas alteraciones.

```python
def my_decorator(func):
    def wrapper():
        print("Something is happening before the function is called.")
        func()
        print("Something is happening after the function is called.")
    return wrapper


def say_whee():
    print("Whee!")


say_whee = my_decorator(say_whee)

print(say_whee)
print('------------')
say_whee()
```

```
<function my_decorator.<locals>.wrapper at 0x7fc54d179d08>
------------
Something is happening before the function is called.
Whee!
Something is happening after the function is called.
```

¿A qué equivale un decorador en Python?

¿Qué es un decorador en Python?

En Python un **decorador** es un **callable** que recibe como argumento otro **callable** (que es **decorado**) y devuelve otro **callable** (**wrapper**) que en su implementación invoca al **callable** que está siendo **decorado.**

```python
def my_decorator(func):
    def wrapper():
        print(f'{func.__name__}:')
        func()
    return wrapper


@my_decorator
def decorated_function():
    print('Hi!')


print(decorated_function)
print('----------------')
decorated_function()
```


```
<function my_decorator.<locals>.wrapper at 0x7f9ade9f6d08>
----------------
decorated_function:
Hi!
```

```python
def do_twice(func):
    def wrapper(*args, **kwargs):  # capture any positional or keyword argument
        func(*args, **kwargs)  # unpack arguments
        func(*args, **kwargs)
    return wrapper


@do_twice
def print_smile():
    print(':D')


@do_twice
def greet(name):
    print(f'Hi {name}!')


print_smile()
print('-----')
greet('World')
```

```
:D
:D
-----
Hi World!
Hi World!
```

Decorando funciones con argumentos

Devolviendo los valores de funciones decoradas

El **decorador** decide que acción realizar con el valor devuelto por el **callable** que está siendo **decorado.** Puede no devolverlo, devolverlo sin modificaciones, devolver una versión modificada o devolver cualquier otro valor.

```
def count_arguments(func):
    def wrapper(*args, **kwargs):
        print(f'Number of arguments: {len(args) + len(kwargs)}')
        # return the value returned by func
        # without modifications
        return func(*args, **kwargs)
    return wrapper
```

Crisis de identidad (I)

```python
def my_decorator(func):
    def wrapper(*args, **kwargs):
        """Decorator docstring."""
        print(f'Calling {func.__name__}')
        return func(*args, **kwargs)
    return wrapper


@my_decorator
def greet(name):
    """Say hi to someone."""
    print(f'Hi {name}!')


print(greet)
print(greet.__name__)
print(greet.__doc__)
```

```
<function my_decorator.<locals>.wrapper at 0x7f17b20bfd08>
wrapper
Decorator docstring.
```

Al **decorar** un **callable** su referencia cambia y apunta al **wrapper** del **decorador.**

Crisis de identidad (II)

El **decorador** `@functools.wraps` cambia algunos atributos especiales del **callable** devuelto por el **decorador** para que reflejen los del **callable** que está siendo **decorado.**

```
<function greet at 0x7fc941bc56a8>
greet
Say hi to someone.
```

```python
import functools


def my_decorator(func):
    @functools.wraps(func)
    def wrapper(*args, **kwargs):
        """Decorator docstring."""
        print(f'Calling {func.__name__}')
        return func(*args, **kwargs)
    return wrapper


@my_decorator
def greet(name):
    """Say hi to someone."""
    print(f'Hi {name}!')


print(greet)
print(greet.__name__)
print(greet.__doc__)
```

Decoradores anidados

Se pueden aplicar varios **decoradores** al mismo **callable.**

```
<a>
<b>
Hi World!
</b>
</a>
```

```python
def decorator_a(func):
    def wrapper(*args, **kwargs):
        print('<a>')
        func(*args, **kwargs)
        print('</a>')
    return wrapper


def decorator_b(func):
    def wrapper(*args, **kwargs):
        print('<b>')
        func(*args, **kwargs)
        print('</b>')
    return wrapper


@decorator_a
@decorator_b
def greet(name):
    print(f'Hi {name}!')


greet('World')
```

Decoradores con argumentos

Se pueden definir **decoradores** que acepten argumentos.

En caso de hacerlo se necesita definir un **callable** que devuelva un **decorador.**

```
Hello World
Hello World
Hello World
```

```python
import functools


def repeat(num_times):
    def decorator_repeat(func):
        @functools.wraps(func)
        def wrapper_repeat(*args, **kwargs):
            for _ in range(num_times):
                value = func(*args, **kwargs)
            return value
        return wrapper_repeat
    return decorator_repeat


@repeat(num_times=3)
def greet(name):
    print(f"Hello {name}")


greet("World")
```

Decoradores con argumentos opcionales

```python
import functools


def repeat(_func=None, *, num_times=2):
    def decorator_repeat(func):
        @functools.wraps(func)
        def wrapper_repeat(*args, **kwargs):
            for _ in range(num_times):
                value = func(*args, **kwargs)
            return value
        return wrapper_repeat

    if _func is None:
        return decorator_repeat
    else:
        return decorator_repeat(_func)
```

```python
@repeat
def say_whee():
    print("Whee!")


@repeat(num_times=3)
def greet(name):
    print(f"Hello {name}")


say_whee()
greet("World")
```

```
Whee!
Whee!
Hello World
Hello World
Hello World
```

Decorando métodos

Se pueden utilizar **decoradores** para los métodos definidos dentro de una clase.

Algunos **decoradores** importantes son:

- @classmethod
- @staticmethod
- @property (se utiliza para construir getters y setters)

```
class C:
    def __init__(self):
        self._x = None

    @property
    def x(self):
        """I'm the 'x' property."""
        return self._x

    @x.setter
    def x(self, value):
        self._x = value

    @x.deleter
    def x(self):
        del self._x
```

Decorando clases

Se puede escribir un **decorador** para clases de la misma forma que un **decorador** para funciones.

La diferencia es que cuando se usa un **decorador** en una clase, el **decorador** recibe esa clase como argumento.

```python
import functools


def singleton(cls):
    """Make a class a Singleton class (only one instance)"""
    @functools.wraps(cls)
    def wrapper_singleton(*args, **kwargs):
        if not wrapper_singleton.instance:
            wrapper_singleton.instance = cls(*args, **kwargs)
        return wrapper_singleton.instance
    wrapper_singleton.instance = None
    return wrapper_singleton


@singleton
class TheOne:
    pass
```

Clases decoradoras

Si se desea mantener el estado es más claro utilizar una **clase decoradora.**

El método `__call__()` es llamado cuando una instancia de una clase es invocada como **callable.**

```python
import functools


class CountCalls:

    def __init__(self, func):
        functools.update_wrapper(self, func)
        self._func = func
        self._num_calls = 0

    def __call__(self, *args, **kwargs):
        self._num_calls += 1
        print(f"Call {self._num_calls} of {self._func.__name__}")
        return self._func(*args, **kwargs)


@CountCalls
def say_whee():
    print("Whee!")


say_whee()
say_whee()
```

```
Call 1 of say_whee
Whee!
Call 2 of say_whee
Whee!
```

code()

Ejercicios 10

- Crear un decorador `@debug` que imprima el nombre de una función, los argumentos con los que fue llamada y su valor de retorno.
- Crear un decorador `@validate_types` que reciba un argumento opcional `valid_type`. El decorador debe lanzar una excepción `TypeError` cuando alguno de los argumentos de la función decorada no sea una instancia de `valid_type`. Si `valid_type` no es provisto entonces el decorador validará que ningún argumento sea `None`. Se debe poder especificar más de un tipo válido con una tupla de tipos.

Bibliografía

- Python docs
- PythonDecorators
- Primer on Python Decorators
- Advanced Uses of Decorators