



códigofacilito



# Decoradores

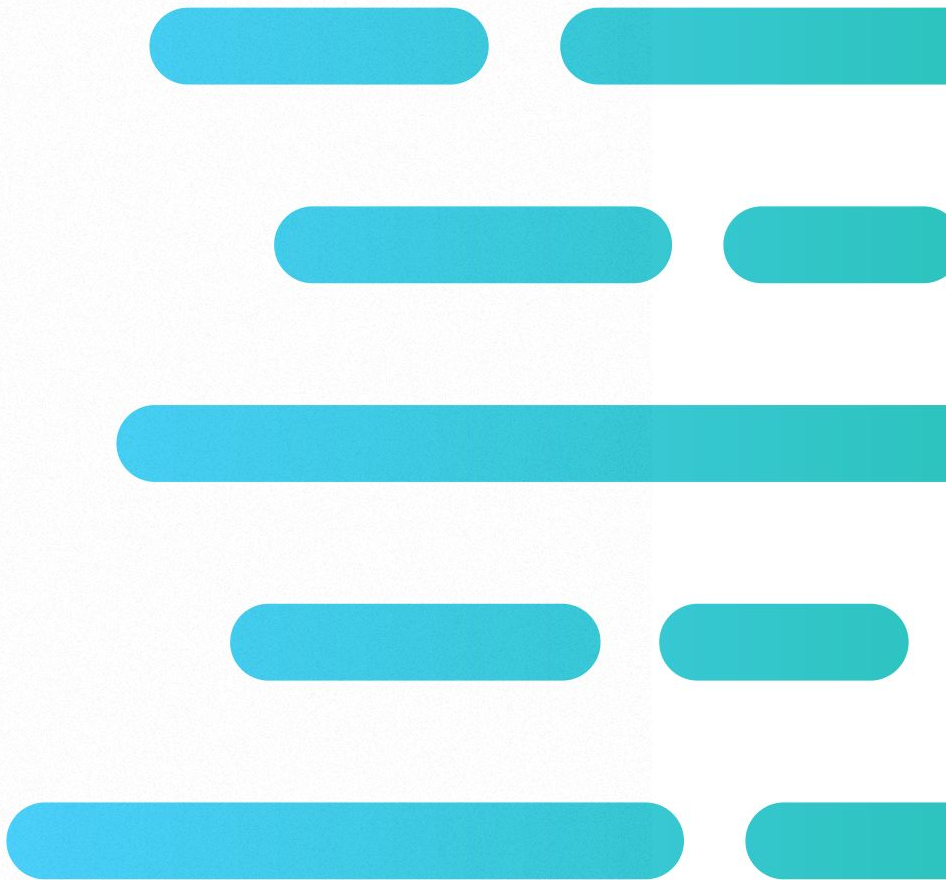
Bootcamp Python Avanzado

Carolina - Backend Developer





Código de la clase 🦆





- >\_ ¿Qué son los decoradores?
- >\_ Cómo trabajan las funciones
- >\_ Creando mi primer decorador
- >\_ Resumen





¿Qué son los  
decoradores?







>\_

**Son un patrón de diseño en Python que permite agregar funcionalidades a un objeto existente (funciones o clases) sin modificar su estructura.**



Recurso recomendado:

<https://www.datacamp.com/tutorial/decorators-python>



```
@example_decorator  
def test_function()  
    return "output"
```



# Cómo trabajan las funciones





>\_

**Las funciones son muy importantes en Python y estas retornan un valor de acuerdo a los argumentos que les pasamos**



```
def plus_one(number):  
    return number + 1
```



Recurso recomendado:

<https://realpython.com/primer-on-python-decorators/#simple-decorators>





>\_

**Podemos asignar el valor  
retornado por una función  
a una variable y llamar la  
variable directamente**



```
def plus_one(number):  
    return number + 1  
  
add_one = plus_one  
add_one(5)
```







>\_

## Podemos definir funciones dentro de otras funciones (closures)

```
def plus_one(number):  
    def add_one(number):  
        print("Executing add_one")  
        return number + 1  
  
    print("Executing plus_one")  
    result = add_one(number)  
    return result  
  
plus_one(4)
```



Recurso recomendado:

<https://codigofacilito.com/articulos/closures-python>



>\_

**Podemos pasar funciones  
como argumentos de  
otras funciones**

```
def plus_one(number):  
    print("Executing plus_one")  
    return number + 1  
  
def function_call(function):  
    print("Executing function_call")  
    number_to_add = 5  
    return function(number_to_add)  
  
function_call(plus_one)
```





>\_

**Una función puede  
retornar otra función**



```
def hello_function():  
    def say_hi():  
        print("Executing say_hi")  
        return "Hi"  
    print("Executing hello_function")  
    return say_hi
```

```
hello = hello_function()  
hello()
```







**Las funciones anidadas  
tienen acceso a las  
variables de la función  
envolvente**



```
def print_message(message):  
    """Enclosing Function"""  
  
    def message_sender():  
        """Nested Function"""  
  
        print(message)  
  
    message_sender()  
  
print_message("Some random message")
```



# Creando mi primer decorador





```
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()
```







## Decoradores de propósito general



```
def a_decorator_passing_arbitrary_arguments(function_to_decorate):  
    def a_wrapper_accepting_arbitrary_arguments(*args,**kwargs):  
        print('The positional arguments are:', args)  
        print('The keyword arguments are:', kwargs)  
        function_to_decorate(*args, **kwargs)  
    return a_wrapper_accepting_arbitrary_arguments  
  
@a_decorator_passing_arbitrary_arguments  
def function_with_arguments(a, b, c):  
    print(a, b, c)  
  
function_with_arguments(1,2,3)
```



**\*args:** es usado para recibir múltiples argumentos por posición.

**\*\*kwargs:** es usado para recibir múltiples argumentos por medio de una llave o key.



Recurso recomendado:

[https://python-intermedio.readthedocs.io/es/latest/args\\_and\\_kwargs.html](https://python-intermedio.readthedocs.io/es/latest/args_and_kwargs.html)

<https://docs.python.org/es/3/tutorial/controlflow.html#more-on-defining-functions>

```
def test_args_and_kwargs(*args, **kwargs):
    print("Showing *args:")
    for arg in args:
        print("Arguments of *args:", arg)
    print("Showing **kwargs:")
    for key, value in kwargs.items():
        print(f"{key} = {value}")

args = (1, 2, "Hola")
kwargs = {"first_name": "Carolina", "last_name": "Gomez"}
test_args_and_kwargs(args, **kwargs)

----
Showing *args:
Arguments of *args: (1, 2, 'Hola')
Showing **kwargs:
first_name = Carolina
last_name = Gomez
```



## Debugueando Decoradores

```
# importing the module
import functools

# decorator
def make_geek_happy(func):
    @functools.wraps(func)
    def wrapper():
        ...
```



Recurso recomendado:

<https://www.geeksforgeeks.org/debugging-decorators-in-python/>





# Resumen





Los decoradores alteran dinámicamente una función, método o clase sin tener que cambiar el código de estas para ser decoradas. Usar decoradores en Python ayuda a que tu código sea DRY (Don't Repeat Yourself). Algunos casos de uso son:

- Autorización en frameworks como Flask y Django
- Logging
- Medición de ejecución de tiempo
- Sincronización

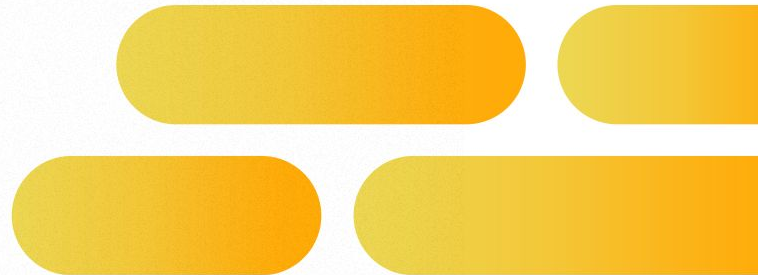


Recurso recomendado:

<https://python-intermedio.readthedocs.io/es/latest/decorators.html>



# ¿Preguntas?



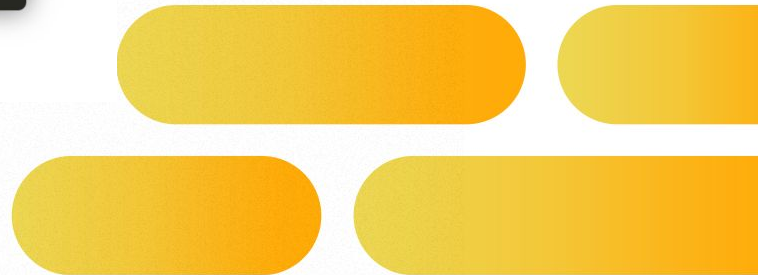




# Tarea:

Crea un decorador que valide si las entradas de la siguiente función son enteros, si no lo son retornar un `TypeError`.

```
def add(a, b):  
    return a + b
```





# ¡Gracias!

