



Conceptos de Python para el Backend

Bootcamp Backend con Python

Carolina - Backend Developer





- **Senior Backend Developer Gorilla Logic**
- **Co organizadora Python Colombia y Django Girls Colombia**
- **Amante del mar y deportes acuáticos.**
-  **/carolinagomezt**





Código de la clase 🦆





- >_ Decoradores
- >_ Funciones Lambda
- >_ Properties
- >_ Métodos de instancia, métodos de clase y métodos estáticos





> Decoradores





¿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://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>



>_ Funciones Lambda





¿Qué son las funciones lambda?





>_

Son también conocidas como funciones anónimas y son funciones que pueden definir cualquier número de parámetros pero una única expresión. Esta expresión es evaluada y devuelta.



```
# lambda parámetros: expresión  
cuadrado = lambda x: x ** 2  
cuadrado(4)  
>> 16
```



Recurso recomendado:

<https://j2logo.com/python/funciones-lambda-en-python/>



Uso apropiado de las funciones lambda





>_

Algunas veces las funciones lambda no son muy recomendadas por que la lectura de su código a veces es complicada e imponen un pensamiento funcional, y la sintaxis puede ser un poco enredada, por eso se aconseja usarla en casos específicos y con otras funciones como:

- **map()**
- **filter()**
- **reduce()**
- **sorted()**



Recurso recomendado:

<https://realpython.com/python-lambda/>



>_ Properties





¿Qué son las properties?





Las properties son la forma pythonica de evitar la creación de métodos para obtener y modificar atributos de una clase. Esta función nos ayuda a convertir atributos de una clase en properties o managed attributes.



Recurso recomendado:

<https://realpython.com/python-property/>

```
class Circle:
    def __init__(self, radius):
        self._radius = radius

    @property
    def radius(self):
        """The radius property."""
        print("Get radius")
        return self._radius

    @radius.setter
    def radius(self, value):
        print("Set radius")
        self._radius = value

    @radius.deleter
    def radius(self):
        print("Delete radius")
        del self._radius
```




>_ Métodos de instancia, métodos de clase y métodos estáticos





>_

Los **métodos de instancia** son creados para modificar un objeto instanciado de una clase.

Los **métodos de clase** trabajan directamente con la clase, desde que su parámetro es la clase en sí.

Los **métodos estáticos** no saben nada acerca de la clase, solo trabajan con los parámetros recibidos.



Recurso recomendado:

<https://realpython.com/instance-class-and-static-methods-demystified/>



>_

```
class MyClass:
    # Ejemplo método de instancia
    def method(self):
        return 'instance method called', self

    # Ejemplo método de clase
    @classmethod
    def class_method(cls):
        return 'class method called', cls

    #Ejemplo método estático
    @staticmethod
    def static_method():
        return 'static method called'
```



Recurso recomendado:

<https://www.programiz.com/python-programming/methods/built-in/classmethod>



Resumen





>_

- Los métodos de instancia necesitan una instancia de una clase y pueden acceder dicha instancia por medio de self.
- Los métodos de clase no necesitan una instancia de una clase. Ellos no pueden acceder a la instancia (self), pero tienen acceso a la clase por medio de cls.
- Los métodos estáticos no tienen acceso a cls o self. Ellos trabajan como funciones regulares pero pertenecen al namespace de la clase.
- Los métodos estáticos y de clase son útiles al momento de diseñar una clase, estos tienen muchos beneficios en el mantenimiento y lectura del código.

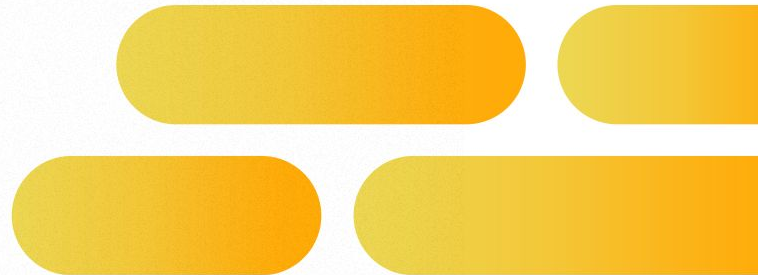


Recurso recomendado:

<https://docs.python.org/3/tutorial/classes.html>



¿Preguntas?

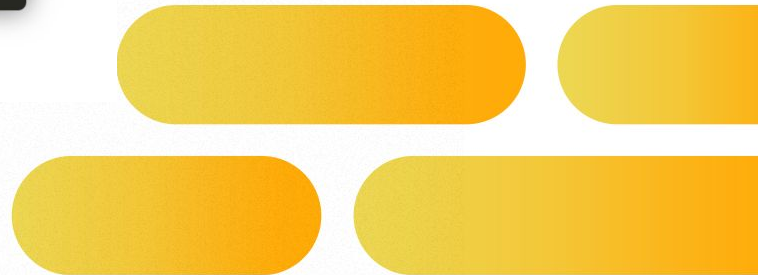




Tarea:

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!

