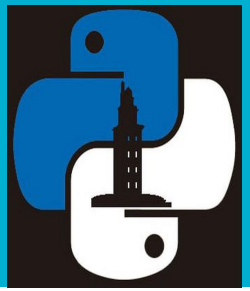
PYTHON CORUÑA – TECH TALK



Decoradores en Python

Tech-talk

Decoradores

Definición en una frase:

Son funciones que actúan como wrappers y permiten modificar el comportamiento de otras funciones

- Para entenderlo son necesarios 3+1 conceptos sencillos expuestos a continuación
- La sintaxis tampoco es trivial, y hay que practicarla
- Es de los primeros conceptos de Python a nivel medio-avanzado

Decoradores Conceptos básicos - Function objects

- Cuando definimos una función en python, obtenemos un objeto *function*

- Los objetos *function* son *callable, es decir,* puedes llamarlos añadiendo un paréntesis
  Cuando no usas paréntesis, es una variable que hace referencia a la función

- Todas las funciones en Python son Callable, pero no todos los callable son funciones

```
>>> def greet(name):
...     print("Hi", name)
...
>>> greet
<function greet at 0x7f61693c5940>
>>> greet("Trey")
Hi Trey
```

Decoradores: Conceptos básicos - First-class objects

- Las funciones en Python son objetos de primera clase (first-class objects)

- Esto quiere decir que pueden ser pasados como argumentos a otras funciones igual que un int, str, float etc

Decoradores: Conceptos básicos - First-class objects

```python
from typing import Callable
from typing import Optional


def argument_function_one() -> None:
    print(f'{"I am a function passed as an argument"}')


def argument_function_two(name: str) -> None:
    print(f"I am a function passed as an argument wich receives the argument {name}")


def receive_func_as_arg(func: Callable, proxy_arg: Optional[str] = None) -> None:
    print(f"I receive {func} as an argument")

    func(proxy_arg) if proxy_arg else func()


argument_function_one()
argument_function_two("demo")
receive_func_as_arg(argument_function_one)
receive_func_as_arg(argument_function_two, proxy_arg="pedro")
```

Sin arg de entrada, sin retorno y hace print()

Con arg de entrada, sin retorno y hace print()

Recibe una función como arg y la ejecuta dentro

```
I receive <function argument_function_one at 0x7fb9c1213d90> ...
I am a function passed as an argument

I receive <function argument_function_two at 0x7fb9c1135f30>...
I am a function passed as an argument which receives the argument pedro
```

Decoradores: conceptos básicos - inner functions

Inner functions - Es posible definir funciones dentro de otras funciones

- El orden de declaración de las funciones no importa
- Las funciones ni siquiera existen hasta que se ejecuta la función que las contiene, tienen un scope local.

Decoradores: conceptos básicos - inner functions

funcion que tiene definidas 2 funciones internas y las ejecuta

```python
def parent_function():
    print("Printing from the parent() function")

    def inner_function_one() -> None:
        print("Printing from the first inner function")

    def inner_function_two() -> None:
        print("Printing from the second inner function")

    inner_function_two()
    inner_function_one()


parent_function()
print("\n")
try:
    inner_function_one()
except NameError:
    print(f'{"NameError occurred. The function is not defined."}')

```

Inner function

Inner function

```
Printing from the parent() function
Printing from the second inner function
Printing from the first inner function
```

La función interna no existe fuera del ámbito de parent_function()

```
NameError occurred. The function is not defined.
```

Decoradores: conceptos básicos - return functions

Las funciones pueden devolver otras funciones

```python
from typing import Callable


def choose_your_return_function(number: int) -> Callable:
    def function_one() -> None:
        print(f'{"hi I am function one"}')

    def function_two() -> None:
        print(f'{"hi I am function two"}')

    match number:
        case 1:
            return function_one
        case 2:
            return function_two
        case _:
            print(f'{"No return for wildcard"}')


first = choose_your_return_function(1)
print(first)
first()
```

Optional[Callable]

función que tiene definidas 2 funciones internas y las ejecuta

La variable **first** contiene ahora una referencia a una función
Es *__callable__* sólo añadiendo paréntesis

```
<function choose_your_return_function.<locals>.function_one at 0x7f912ffd9f30>
hi I am function one
```

Decoración simple - Ejemplo

```python
from typing import Callable


def my_decorator(func: Callable) -> None:
    def wrapper() -> None:
        HOUR = 22
        if 7 <= HOUR < 22:
            func()
        else:
            print("shhh")

    return wrapper


def shout() -> None:
    print(f'{"HOLAAAAA!!!"}')


shout()
print(shout)

shout = my_decorator(shout)
print(shout)
shout()
```

First-clas object

Callable

Inner function

Return function

Aquí shout contiene una referencia a la función L15

Aquí shout contiene una referencia a una inner function llamada wrapper, Dentro de my_decorator y que contiene una referencia a la función shout original

Hemos modificado el comportamiento de shout a través de otra función !!

```
HOLAAAAA!!!
<function shout at 0x7fec44981f30>
<function my_decorator.<locals>.wrapper at 0x7fec44862cb0>
shhh
```

Sintaxis en Python - Syntactic sugar I

```python
from typing import Callable


def my_decorator(func: Callable) -> None:
    def wrapper() -> None:
        HOUR = 22
        if 7 <= HOUR < 22:
            func()
        else:
            print("shhh")

    return wrapper


def shout() -> None:
    print(f'{"HOLAAAAA!!!"}')


shout = my_decorator(shout)
shout()
```

Python facilita el último paso del procedimiento anterior internamente permitiendo decorar una función sólo con @

```python
def my_decorator(func: Callable) -> None:
    def wrapper() -> None:
        HOUR = 22
        if 7 <= HOUR < 22:
            func()
        else:
            print("shhh")

    return wrapper


@my_decorator
def shout() -> None:
    print(f'{"HOLAAAAA!!!"}')
```

Sintaxis en Python - Syntactic sugar II

```python
from typing import Callable


def my_decorator(func: Callable) -> None:
    def wrapper() -> None:
        HOUR = 22
        if 7 <= HOUR < 22:
            func()
        else:
            print("shhh")

    return wrapper


@my_decorator
def shout() -> None:
    print(f'{"HOLAAAAA!!!"}')


print(shout)
shout()
```

```
<function my_decorator.<locals>.wrapper at
0x7f5761152cb0>
shhh
```

```python
import functools
from typing import Callable


def my_decorator(func: Callable) -> None:
    @functools.wraps(func)
    def wrapper() -> None:
        HOUR = 22
        if 7 <= HOUR < 22:
            func()
        else:
            print("shhh")

    return wrapper


@my_decorator
def shout() -> None:
    print(f'{"HOLAAAAA!!!"}')

print(shout)
```

```
<function shout at 0x7f154400ecb0>
```

Valores de retorno - Ejemplo

Escribe un decorador que cambie el retorno de las funciones a mayúscula si el total de caracteres es impar, en caso contrario devolverá lo mismo que la función original, además de devolver dicho número

- Paso 1: Escribir las funciones a decorar
- Paso 2: Devolver un valor modificado desde el wrapper
- Paso 3: Devolver el valor original desdel el wrapper
- Paso 4: Hacer que las funciones a decorar tengan un número variable de argumentos
- Paso 5: Modificar el decorador para que sea capaz de decorar ambas

Qué ejecutamos al ejecutar una función decorada?
↠ la función wrapper, ella es la que controla qué se devuelve

Para que wrapper valga para cualquier número de argumentos de entrada, tenemos que usa la firma proxy_args

Valores de retorno - Ejemplo

```python
import functools
from typing import Callable
from typing import Tuple


def conditional_uppercase(func: Callable) -> Callable:
    @functools.wraps(func)
    def conditional_uppercase_wrapper(*args, **kwargs) -> Tuple[str,int]:
        original_value = func(*args, **kwargs)
        lenght = len(original_value)

        if lenght %2 ==0:
            return original_value, lenght
        else:
            return original_value.upper(), lenght

    return conditional_uppercase_wrapper

@conditional_uppercase
def test_return_even_str() -> str:
    return f'{"this is even"}'
```

Firma genérica

La función que ejecutamos es wrapper,
Por lo cual es la que maneja el return

Decoradores que aceptan argumentos - Ejemplo

A veces necesitamos que un decorador se comporte de una u otra forma en función de un parámetro de entrada

- Al igual que el parámetro func estará disponible en el scope de wrapper, si subimos un nivel el parámetro de entrada estará disponible para el decorador

Decoradores que aceptan argumentos - Ejemplo

```python
import functools
from typing import Callable
from typing import Tuple


def decorator_factory(input_arg: int) -> Callable:

    def conditional_uppercase(func: Callable) -> Callable:
        @functools.wraps(func)
        def conditional_uppercase_wrapper(*args, **kwargs) -> Tuple[str,int]:
            original_value = func(*args, **kwargs)
            lenght = len(original_value)

            if input_arg %2 ==0:
                return original_value, lenght
            else:
                return original_value.upper(), lenght

        return conditional_uppercase_wrapper

    return conditional_uppercase

@decorator_factory(2)
def test_return_even_str() -> str:
    return f'{"this is even"}'
```

Decoradoradores y OOP

- Podemos hacer que el comportamiento de un decorador depende de su propio estado, implementando una clase decoradora

- Se pueden decorar métodos de instancia o clases enteras

Decoradores con estado - clase decoradora

```python
import functools


class MyDecorator:
    def __init__(self, func):
        functools.update_wrapper(self, func)
        self.func = func
        self.num_times = 0

    def __call__(self, *args, **kwargs):
        original_value = self.func(*args, **kwargs)
        self.num_times += 1
        print(f"Call {self.num_times} of {self.func.__name__!r}")

        if self.num_times % 2 == 0:
            return original_value
        else:
            return original_value.upper()


@MyDecorator
def test_return_even_str() -> str:
    return f'{"this is even"}'


@MyDecorator
def test_return_odd_str() -> str:
    return f'{"this is odd"}'
```

Decoradores - Decorando un método de instancia

```python
def conditional_uppercase(func: Callable) -> Callable:
    @functools.wraps(func)
    def conditional_uppercase_wrapper(*args, **kwargs) -> Tuple[str, int]:
        original_value = func(*args, **kwargs)
        lenght = len(original_value)

        if lenght % 2 == 0:
            return original_value, lenght
        else:
            return original_value.upper(), lenght

    return conditional_uppercase_wrapper


class Cadena:
    def __init__(self, initial_value="aa"):
        self.cadena = initial_value

    @conditional_uppercase
    def get_string(self):
        return f"{self.cadena}"

    def add_more_cadena(self, adding_chars):
        self.cadena += adding_chars
```

Referencias

Explicación paso a paso:

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

Libro publicado por *RealPython* (está 'disponible' en pdf)

- *Python tricks: the book* , Dan Bader

Otra buena explicación en inglés

- https://gist.github.com/lnhote/7875074

Katas de decoradores:

- https://www.codewars.com/kata/search/my-languages?q=&tags=Decorator&order_by=sort_date%20desc