


15 Decoradores.

Objetivo. ...

Funciones de Python: ...

[MACTI-Algebra_Lineal_01](#) by [Luis M. de la Cruz](#) is licensed under [Attribution-ShareAlike 4.0 International](#) 

16 Definición.

- Se denomina decorador a la persona dedicada a diseñar el interior de oficinas, viviendas o establecimientos comerciales con criterios estéticos y funcionales.
- En Python, un decorador es una función para modificar otra función.
 - Recibe una función.
 - Regresa otra función.
- Los decoradores son herramientas bonitas y útiles de Python.

16.1 Ejemplo 1.

La función `print_hello()` imprime `Hola mundo pythonico`.

```
def print_hello():  
    print('{:^30}'.format('Hola mundo pythonico'))
```

Crear un decorador que agregue colores al mensaje.

```
def print_hello():  
    print('{:^30}'.format('Hola mundo pythonico'))
```

```
# Uso normal de la función  
print_hello()
```

Hola mundo pythonico

```
from colorama import Fore, Back, Style  
  
# Decorador  
def mi_decorador1(f):  
  
    # La función que hace el decorado.
```

```
def envoltura():
    linea = '-' * 30
    print(Fore.BLUE + linea + Style.RESET_ALL)
    print(Back.GREEN + Fore.WHITE, end='')

    f() # Ejecución de la función

    print(Style.RESET_ALL, end='')
    print(Fore.BLUE + linea + Style.RESET_ALL)

# Regresamos la función decorada
return envoltura

# Decorando la función.
print_hello_colored = mi_decorador1(print_hello) # Funcion decorada

# Ahora se ejecuta la función decorada.
print_hello_colored()
```

```
-----
Hola mundo pythonico
-----
```

16.2 Ejemplo 2.

La función `print_message(m)` imprime el mensaje que recibe como parámetro.

```
def print_message(m):
    print('{:^30}'.format(m))
```

Modificar el decorador creado en el ejemplo 1 para que se pueda recibir el parámetro `m`.

```
# Decorador
def mi_decorador2(f):

    # La función que hace el decorado.
    # Ahora recibe un parámetro
    def envoltura(m):
        linea = '-' * 30
        print(Fore.BLUE + linea + Style.RESET_ALL)
        print(Back.GREEN + Fore.WHITE, end='')

        f(m) # Ejecución de la función

        print(Style.RESET_ALL, end='')
        print(Fore.BLUE + linea + Style.RESET_ALL)
```

```
# Regresamos la función decorada
return envoltura
```

```
# La función se puede decorar en su definición como sigue
@mi_decorador2
def print_message(m):
    print('{:^30}'.format(m))
```

```
# Entonces se puede usar la función con su nombre original
print_message('bueno, bonito y barato')
```

```
-----
bueno, bonito y barato
-----
```

16.3 Ejemplo 3.

Decorar las funciones `sin()` y `cos()` de la biblioteca `math`.

```
def mi_decorador3(f):

    def coloreado(x):

        # Construimos una cadena coloreada con el
        # resultado de la evaluación de f(x)
        res = Fore.GREEN + f.__name__
        res += '(' + Style.BRIGHT + str(x) + Style.RESET_ALL + Fore.GREEN + ')' =
        res += f'{f(x)}'

        # Imprimimos el resultado
        linea = '-' * 80
        print(Fore.BLUE + linea + Style.RESET_ALL)
        print('{:^80}'.format(res))
        print(Fore.BLUE + linea + Style.RESET_ALL)

    return coloreado

from math import sin, cos

sin = mi_decorador3(sin)
cos = mi_decorador3(cos)

for f in [sin, cos]:
    f(3.141596)
```

```
sin(3.141596) = -3.3464102065883993e-06
```

```
cos(3.141596) = -0.99999999999944008
```

16.4 Ejemplo 4.

Decorar funciones con un número variable de argumentos.

```
from random import random, randint, choice, choices

def mi_decorador4(f):
    def envoltura(*args, **kwargs):

        # Construimos una cadena coloreada con el
        # resultado de la evaluación de f(x)
        res = Fore.GREEN + f.__name__
        res += '(' + Style.BRIGHT + f'{args},{kwargs}' + Style.RESET_ALL + Fore.C
        res += f'{f(*args, **kwargs)}'

        # Imprimimos el resultado
        linea = '-' * 80
        print(Fore.BLUE + linea + Style.RESET_ALL)
        print('{:^80}'.format(res))
        print(Fore.BLUE + linea + Style.RESET_ALL)

    return envoltura

random = mi_decorador4(random)
randint = mi_decorador4(randint)
choice = mi_decorador4(choice)
choices = mi_decorador4(choices)

random()
randint(3, 8)
choice([4, 5, 6])

p = [x for x in range(10)]
choices(p, k=3)
```

```
random((),{}) = 0.4390656899525458
```

```
randint((3, 8),{}) = 3
```

```
choice([4, 5, 6], {}, {}) = 5
```

```
choices([0, 1, 2, 3, 4, 5, 6, 7, 8, 9], {}, {'k': 3}) = [5, 3, 9]
```

16.5 Ejemplo 5.

Crear un decorador que calcule el tiempo de ejecución de una función.

```
import time

def crono(f):
    """
    Regresa el tiempo que toma en ejecutarse la funcion.
    """
    def tiempo():
        t1 = time.perf_counter()
        f()
        t2 = time.perf_counter()
        return 'Elapsed time: ' + str((t2 - t1)) + "\n"
    return tiempo

@crono
def miFuncion():
    numeros = []
    for num in (range(0, 10000)):
        numeros.append(num)
    print('\nLa suma es: ' + str((sum(numeros))))

print(miFuncion())
```

La suma es: 49995000

Elapsed time: 0.0012546591460704803

16.6 Ejemplo 6.

Detener la ejecución por un tiempo antes que una función sea ejecutada.

```

from time import sleep

def sleepDecorador(function):

    def duerme(*args, **kwargs):
        sleep(1)
        return function(*args, **kwargs)
    return duerme

@sleepDecorador
def imprimeNumero(num):
    return num

for num in range(1, 6):
    print(imprimeNumero(num), end = ' ')

print('\n --> happy finish!')

```

```

1 2 3 4 5
--> happy finish!

```

16.7 Ejemplo 7.

Crear un decorador que cheque que el argumento de una función que calcula el factorial, sea un entero positivo.

```

def checaArgumento(f):
    def checador(x):
        if type(x) == int and x > 0:
            return f(x)
        else:
            raise Exception("El argumento no es un entero positivo")
    return checador

@checaArgumento
def factorial(n):
    if n == 1:
        return 1
    else:
        return n * factorial(n-1)

for i in range(1,10):
    print(i, factorial(i))

```

```
1 1
2 2
3 6
4 24
5 120
6 720
7 5040
8 40320
9 362880
```

```
print(factorial(-1))
```

Exception: El argumento no es un entero positivo

16.8 Ejemplo 8.

Contar el número de llamadas de una función.

```
def contadorDeLlamadas(func):

    def cuenta(*args, **kwargs):
        cuenta.calls += 1
        return func(*args, **kwargs)

    # Variable estática que lleva la cuenta
    cuenta.calls = 0

    return cuenta

@contadorDeLlamadas
def suma(x):
    return x + 1

@contadorDeLlamadas
def mulp1(x, y=1):
    return x*y + 1

print('Llamadas a suma = {}'.format(suma.calls))

for i in range(4):
    suma(i)

mulp1(1, 2)
mulp1(5)
mulp1(y=2, x=25)
```

```
print('Llamadas a suma = {}'.format(suma.calls))  
print('Llamadas a multp1 = {}'.format(mulp1.calls))
```

```
Llamadas a suma = 0  
Llamadas a suma = 4  
Llamadas a multp1 = 3
```

16.9 Ejemplo 9.

Decorar una función con diferentes saludos.

```
def buenasTardes(func):  
    def saludo(x):  
        print("Hola, buenas tardes, ", end='')  
        func(x)  
    return saludo  
  
def buenosDias(func):  
    def saludo(x):  
        print("Hola, buenos días, ", end='')  
        func(x)  
    return saludo  
  
@buenasTardes  
def mensaje1(hora):  
    print("son las " + hora)  
  
mensaje1("3 pm")  
  
@buenosDias  
def mensaje2(hora):  
    print("son las " + hora)  
  
mensaje2("8 am")
```

```
Hola, buenas tardes, son las 3 pm  
Hola, buenos días, son las 8 am
```

16.10 Ejemplo 10.

El ejemplo anterior se puede realizar como sigue:


```
def saludo(expr):  
    def saludoDecorador(func):  
        def saludoGenerico(x):  
            print(expr, end='')  
            func(x)  
        return saludoGenerico  
    return saludoDecorador  
  
@saludo("Hola, buenas tardes, ")  
def mensaje1(hora):  
    print("son las " + hora)  
  
mensaje1("3 pm")  
  
@saludo("Hola, buenos días, ")  
def mensaje2(hora):  
    print("son las " + hora)  
  
mensaje2("8 am")  
  
@saludo("καλημερα ")  
def mensaje3(hora):  
    print(" <--- en griego " + hora)  
  
mensaje3(" :D ")
```

Hola, buenas tardes, son las 3 pm

Hola, buenos días, son las 8 am

καλημερα <--- en griego :D