Decorador en Python.

ESTUDIANTE: Jasser Esteban Villarreal Buitrago

C.C 1000 782 795

DOCENTE: Santiago Echeverri Arteaga

FACULTAL DE CIENCIAS Y TEGNOLOGIAS

PROGRAMA: Física.

UNIVERCIDAD DEL QUINDIO

2023

decorador en Python

Un decorador en Python es una función que recibe otra función como parámetro, le añade cosas y retorna una función diferente. Son herramientas muy útiles. Nos permiten envolver una función dentro de otra y modificar el comportamiento de esta última sin modificarla permanentemente.

Conceptos en los que se basan los decoradores

Funciones dentro de una función

En el siguiente ejemplo definimos la funcion_intera() dentro la funcion_externa() y hacemos una llamada a la funcion_interna() también dentro de la funcion_externa(). Notar la indentación del código.

```
def funcion_externa():
  print("Código de la funcion_externa()")
    def funcion_interna():
      print("Código de la funcion_interna()")
    funcion_interna()
```

Si ahora llamamos a la funcion_externa() obtenemos el siguiente resultado.

```
>>> funcion_externa()
Código de la funcion_externa()
Código de la funcion_interna()
```

Es decir, que podemos llamar a la funcion_interna() porque lo hacemos dentro de la funcion_externa() que es donde la hemos definido. Efectivamente no podemos llamar a la funcion_externa() desde otros lugares de nuestro código ya que el intérprete de Python nos dice que tal función no existe.

```
>>> funcion_interna()
Traceback (most recent call last):
```

```
  File "<stdin>", line 1, in <module>
NameError: name 'funcion_interna' is not defined
```

Retornando funciones

En el punto anterior acabamos de concluir que la funcion_interna() de nuestro ejemplo sólo se puede llamar dentro de la funcion_externa(). Pero ¿qué pasaría si la funcion_externa() retornara la funcion_interna()? Para saberlo, vamos a modificar un poco el código de nuestra funcion_externa() para reflejar tal cambio.

```
def funcion_externa():
  print("Código de la funcion_externa()")
  def funcion_interna():
    print("Código de la funcion_interna()")
  return funcion_interna
```

Notar que para el retorno en la última línea no usamos paréntesis ya que en este caso no queremos ejecutar la funcion_interna() sino retornarla. Ahora vayamos un paso más allá y asignemos esta función a una variable que denominamos mi_funcion.

```
>>> mi_funcion = funcion_externa()
Código de la funcion_externa()

>>> mi_funcion()
Código de la funcion_interna()
```

Lo que observamos es que ahora, al haber retornado y asignado la funcion_interna() a la variable mi_funcion, podemos ejecutar el código de la funcion_interna() en otras partes de nuestro programa llamando a mi_funcion().

Funciones como argumentos

Una de las cosas que hemos hecho en el apartado anterior es asignar una función a una variable. Por otro lado sabemos que las funciones admiten variables como argumentos. Esto significa que a una función se le puede pasar otra función como argumento. Veámoslo con el siguiente ejemplo.

```
def una_funcion():
```

```
  return "Código de una_funcion()"
def otra_funcion(alguna_funcion):
  print("Código de otra_funcion()")
  print(alguna_funcion())
```

Como vemos ambas funciones son muy simples. Por un lado tenemos una_funcion() que retorna un *string*. Por otro, la función otra_funcion() que imprime un *string*, y a continuación el resultado retornado por la función que se le ha pasado como argumento. Llegados a este punto seguro que ya te imaginas el siguiente paso: ver que sucede cuando le pasamos a otra_funcion() una_funcion() como argumento.

```
>>> otra_funcion(una_funcion)
Código de otra_funcion()
Código de una_funcion()
```

Y lo que vemos es que en otra_funcion() podemos ejecutar sin problema el código de una_funcion().

Estructura de un decorador

Un decorador se compone de las tres ideas que acabamos de ver. Es decir, acepta una función como argumento y define una función interior la cual retorna. Además, dentro de la función interior ejecuta la función que se le ha pasado como argumento. Esto, que puede sonar algo confuso, traducido a código Python queda tal que así:

```
def mi_decorador(funcion_original):
  def funcion_envolvente():
    print("Código antes de la funcion_original()")
    funcion_original()
    print("Código después de la funcion_original()")
  return funcion_envolvente
```

Ahora definamos una función que vamos a pasarle a mi_decorador():

```
def funcion_necesita_decorador():
  print("¡Quiero que me decoren!")
```

Si pasamos la función anterior al decorador definido arriba, guardamos la función retornada por el decorador en una variable y ejecutamos dicha función obtenemos el siguiente resultado:

```
>>> funcion_decorada = mi_decorador(funcion_necesita_decorador)

>>> funcion_decorada()
Código antes de la funcion_original()
¡Quiero que me decoren!
Código después de la funcion_original()
```

Tal y como veíamos en la definición del inicio de ese artículo, en este último paso, hemos añadido funcionalidad extra a una función sin necesidad de añadir dicha funcionalidad explícitamente. En otras palabras, hemos envuelto o "decorado" una función con nueva funcionalidad dejando dicha función intacta, ya que el código que hemos añadido está en el decorador. Si queremos añadir o quitar dicha funcionalidad a la función, basta con añadirle o quitarle el decorador.

En Python la asignación que hemos realizado en la primera línea del código anterior se realiza con el signo @. Es decir, que esa asignación es equivalente a realizar lo siguiente:

```
@mi_decorador
def funcion_necesita_decorador():
  print("¡Quiero que me decoren!")
```

De este modo el código queda muy limpio y fácil de entender. Además, podemos ejecutar la función por su nombre original obteniendo los mismos resultados que antes.

```
>>> funcion_necesita_decorador()

Código antes de la funcion_original()
¡Quiero que me decoren!
Código después de la funcion_original()
```

Decoradores en funciones con parámetros

Para simplificar la explicación del apartado anterior hemos utilizado una función sin parámetros. Sin embargo, es habitual que las funciones tomen parámetros de entrada para realizar alguna operación con ellos. En el siguiente código de ejemplo, al decorar

una función con dos parámetros de entrada con el decorador que hemos definido anteriormente no sucede nada.

```
@mi_decorador
def mostrar_info(nombre, edad):
  print(f"{nombre} tiene {edad} años")
```

Sin embargo, al ejecutar dicha función obtenemos un error. Esto sucede porque la función que hemos definido dentro de mi_decorador no toma argumentos, pero le estamos pasando los dos de la función decorada.

```
>>> mostrar_info("Yoda", 900)
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
TypeError: funcion_envolvente() takes 0 positional arguments but 2 were given
```

Entonces, ¿sólo podemos decorar funciones sin argumentos o tenemos que crear tantas versiones del decorador como números distintos de parámetros de entrada tengan las funciones que vamos a decorar? Pues ni una cosa ni la otra ya que esto lo podemos resolver de un modo más simple. En Python existe el concepto *args, **kwargs que se utiliza para tomar en cuenta números de parámetros variables. Utilizando *args, **kwargs podemos reescribir nuestro decorador del siguiente modo:

```
def mi_decorador(funcion_original):
  def funcion_envolvente(*args, **kwargs):
    print("Código antes de la funcion_original()")
    Funcion_original(*args, **kwargs)
    print("Código después de la funcion_original()")
  return funcion_envolvente
```

Ahora si volvemos a ejecutar la función mostrar_informacion() decorada con la definición anterior de mi_decorador, sí que obtenemos el resultado esperado.

```
>>> mostrar_info("Yoda", 900)
Código antes de la funcion_original()
Yoda tiene 900 años
Código después de la funcion_original()
```

EJEMPLO DE DECORADOR DE PYTHON::

https://github.com/jasservillarreal/tarea.programacion/blob/master/tareas/decorador.py

REFERENCIAS:

https://platzi.com/blog/decoradores-en-python-que-son-y-como-funcionan/#:~:text=Un%20decorador%20en%20Python%20es,esta%20%C3%BAltima%20sin%20modificarla%20permanentemente.

https://www.programaenpython.com/avanzado/decoradores-en-python/