Decoradores en python ¿Qué son) + Python POO.

Índice del contenido:

Decoradores en Python... ¿Qué coño es eso?

Recordemos que una función puede recibir cómo argumento para su parámetro a otra función (función de orden superior o de primer orden), un decorador es precisamente esto, una función (A) que recibe como argumento a otra función (B) y que devuelve una tercera función (C).

¡Importante!

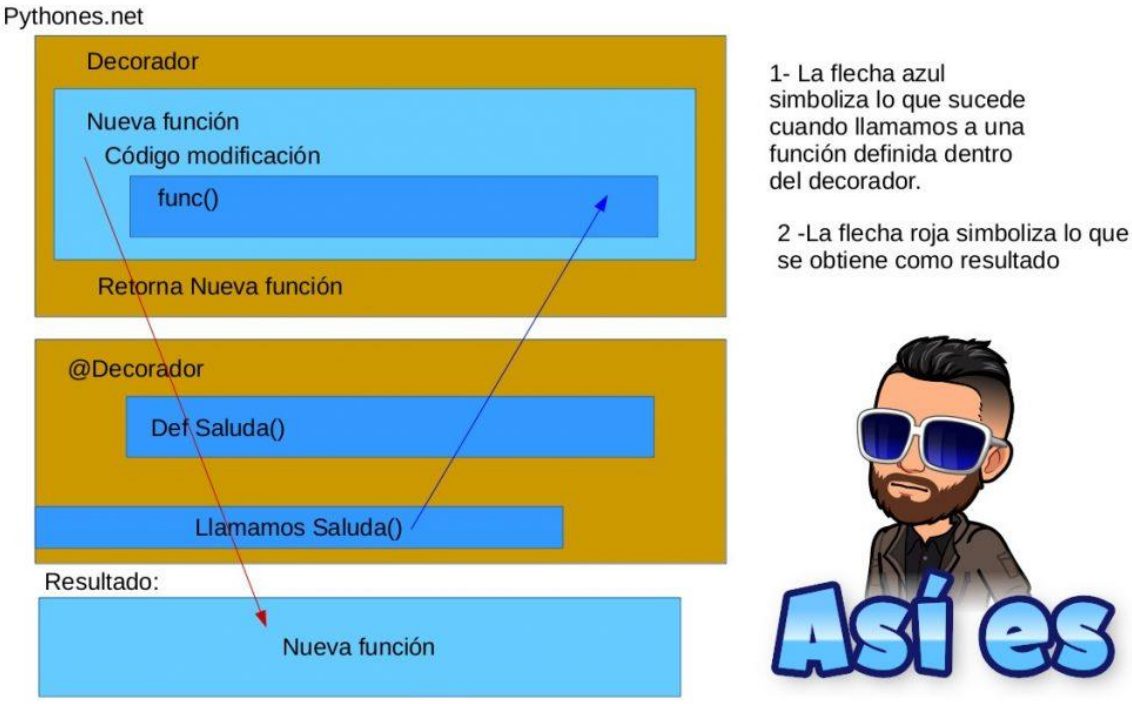
Los decoradores pueden definirse como estereotipos o patrones de diseño. Que permiten a una función (A) o clase de objeto (A) tomar otra función (B) como argumento para devolver una función (C). ¡De esta manera obtendremos funciones dinámicas (que pueden cambiar) sin tener nosotros que cambiar su código fuente!

Un decorador es como un envoltorio con el cual envolvemos una función o una clase.

Podemos dividir a los **decoradores en grupos**, los que permiten argumentos y los que no. Y **los que modifican la firma del método que decoran y los que no**.

Creando un decorador en Python.

```
1.  #!/usr/bin/env python
2.  # -*- coding: utf-8 -*-
3.  # pythones.net
4.  def decorador(func): #Creamos la función decorador (A) con el argumento func
5.      def nueva_funcion(): #Creamos la nueva función (C)
6.          print ("Perro dice:")#Añadimos una modificación a la función (B) dentro de (C)
7.          func() #Aquí estamos incluyendo la función (B) que le dimos como argumento a
    (A)
8.
9.      return nueva_funcion #Para devolver (C)
10.
11. @decorador #Decoramos la función
12. def saluda():
13.     print("Guau!")
14. saluda()
```



Resultado:

Perro dice:

Guau!

Como puedes ver en el diagrama, la función Saluda() pasará por el decorador siempre dando como resultado una nueva función ejecutada que muestra lo que se encuentra dentro del decorador, es decir, la función Saluda() sale decorada!

A sería el decorador

B sería la función saluda

C sería la función nueva.

En este caso func() que brindamos como argumento será la función a la aplicaremos el decorador.

¡¡Bien espero que lo hallas entendido!! Ahora supongamos que definimos **otra función debajo de este decorador**, que en este caso será Despedida ().

```
1.  #!/usr/bin/env python
2.  # -*- coding: utf-8 -*-
3.  # pythones.net
4.  def decorador(func): #Creamos la función decorador (A)
5.      def nueva_funcion(): #Creamos la nueva función (C)
6.          print ("Perro dice:")#Añadimos una modificación a la función (B) dentro (C)
7.          func() #Aquí estamos incluyendo la función (B) que le dimos como argumento a
    (A)
8.                      #Para crear (C)
9.      return nueva_funcion
10. @decorador #Decoramos la función
11. def saluda():
12.     print("Guau!")
13. def despedida():
14.     print ("Chau")
15. saluda()
16.
17. despedida()
```

Resultado:

Perro dice:

Guau!

Chau

Como puedes ver no vuelve a imprimir “Perro dice:” porque no se aplicó el decorador, para eso debe volverse a colocar antes de la función Despedida () así:

```
1.  #!/usr/bin/env python
2.  # -*- coding: utf-8 -*-
3.  # pythones.net
4.  def decorador(func): #Creamos la funcion decorador (A)
5.      def nueva_funcion(): #Creamos la nueva funcion (C)
6.          print ("Perro dice:")#Añadimos una modificacion a la funcion (B) dentro (C)
7.          func() #Aqui estamos incluyendo la funcion (B) que le dimos como argumento a
    (A)
8.
9.      return nueva_funcion #Para crear (C)
10. @decorador #Decoramos la función
11. def saluda(): #Con decorador
12.     print("Guau!")
13. @decorador
14. def despedida(): #Con decorador
15.     print ("Chau")
16. def movercola(): #Sin decorador
17.     print ("El perro mueve la cola")
18. saluda()
19.
20. despedida()
21.
22. movercola()
```

Resultado:

Perro dice:

Guau!

Perro dice:

Chau

El perro mueve la cola

También he agregado una función para que mueva la cola sin decorador para que puedas ver la diferencia.

Decoradores en clases de Python – POO.

Ahora vamos a utilizar esto para trabajar con clases y descubrir que representa un decorador y como lo aplicamos dentro de la programación orientada a objetos. Anteriormente te había mostrado algunos decoradores en variables de métodos de las clases. Pero ahora utilizamos los nuestros propios.

Seguimos con los perritos que ladran, pero en este caso vamos a crear una clase perro y dentro, el método saluda con un mensaje personalizado. Por supuesto vamos a decorar este método utilizando el mismo decorador.

```
1.  #!/usr/bin/env python
2.  # -*- coding: utf-8 -*-
3.  # pythones.net
4.  def decorador(func): # Damos como argumento del decorador a func
5.      def nueva_funcion(self, mensaje): #Aquí debemos colocar los parámetros con los que
    trabaja func
6.          print ("Perro dice:")#Código decorador
7.          func(self, mensaje) #En func agregamos los parámetros con los que trabaja
8.
9.      return nueva_funcion #Retornamos la nueva función
10.
11.
12.  class perro(object): #Creamos la clase heredando de object
13.      def __init__(self, nombre):  #Constructor con el atributo nombre
14.          self.nombre = nombre #Nombre es igual al argumento nombre etc.
15.      @decorador #Aqui antes del método se coloca el decorador!!!
16.      def saluda(self, mensaje): #Metodo saludar del perro, como siempre
17.          self.mensaje = mensaje #El parámetro de saluda mensaje es igual a mensaje
    arg..
18.          print(mensaje) #Imprimir el mensaje ATENCIÓN.
19.          print("Guau!") #Resto del código
20.
21.  maty = perro('Maty') #Instanciamos
22.  maty.saluda('Uso Puppy Linux!') #Cuando llamamos al metodo saluda buscara añadirle el
    decorador
23.  #Osea que se ira hasta arriba. Por ende allí también debimos incluir la instanciacion
    (self) y el
24.  #Argumento mensaje para ambas (nueva_funcion) y func.
```

Resultado:

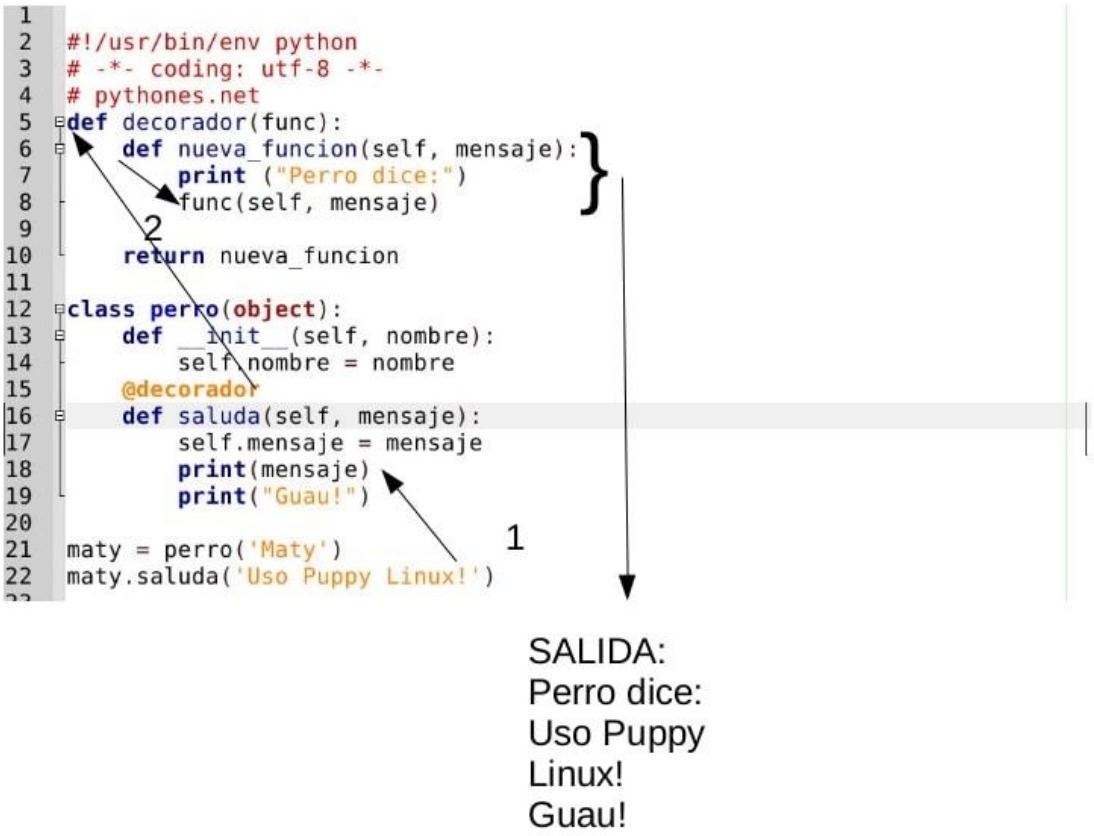
Perro dice:

Uso Puppy Linux!

Guau!

Excelente, pero si aún no lo has entendido lo volvemos a explicar mostrando el orden en el que trabaja el código. Solo para que tengas en cuenta los decoradores son justamente eso, una plantilla, una fábrica de ensamblar una función dentro de otra y devolver una más gorda. Como viste en mi diagrama 😛.

Fíjate como funciona con clases, casi igual solo que superpusimos una clase y nuestra función está dentro como un método de esta clase.



¡Ahora si lo entiendes! ¿Verdad?

Cada vez que se llame al método. saluda() primero se pasará por el decorador y el método saldrá modificado.

La dinámica de los decoradores.

Pero acaso ¿no se suponía que los decoradores nos permitían dinamizar estas funciones?, porque estás utilizando los parámetros en el decorador. ¿Y si usáramos el mismo decorador en otro método? ¿Estamos obligados a usar los mismos parámetros para la función func y nueva_funcion?.

Aquí con utilizar los mismos parámetros nos referimos a que si utilizáramos **el mismo decorador en otro método, el método debería tener el parámetro** mensaje. Por lo tanto, no estamos dinamizando el decorador, solo sería aplicable a ese método. Pero hay una solución para esto:

En este caso fíjate que cambiamos el parámetro “mensaje” por “parametro1” y volvemos a utilizar el método decorado Saluda () y además añadimos otro método llamado Ordeno () que también es decorado con el mismo decorador.

```
1.  #!/usr/bin/env python
2.  # -*- coding: utf-8 -*-
3.  # pythones.net
4.  def decorador(func):
5.      def nueva_funcion(instancia, parametro1):
6.          print ("Perro dice:")
7.          func(instancia, parametro1)
8.
9.      return nueva_funcion
10.
11. class perro(object):
12.     def __init__(self, nombre):
13.         self.nombre = nombre
14.     @decorador
15.     def saluda(self, mensaje):
16.         self.mensaje = mensaje
17.         print(mensaje)
18.         print("Guau!")
19.     @decorador
20.     def ordeno(self, orden):
21.         self.orden = orden
22.         print(orden)
23.         print("La pata, la pata afgsad! Guau!")
24.
25. maty = perro('Maty')
26. maty.saluda('Uso Puppy Linux!')
27. maty.ordeno('Doy la pata')
```

Resultado:

Perro dice:

Uso Puppy Linux!

Guau!

Perro dice:

Doy la pata

La pata, la pata afgsad! Guau!

Como ves cambiamos los parámetros de la función **func ()** mensaje **por parametro1** y self por **instancia.** ¡Agregamos otro método al decorador y lo llamamos también! ¡¡Todo ok, todo perfecto!!

Esta es la estructura correcta para usar decoradores en clases. Así que a partir de ahora recuerda esto a la hora de trabajar con clases porque te permitirá ahorrar mucho código no teniendo que colocar todo el código del decorador en cada uno de los métodos.

Usando clases como decoradores

Ahora te la voy a liar más... Acaso, ¿no notas raro el uso de una función sobre una clase en el código anterior?, es muy desprolijo y estamos modificando un método desde fuera de la clase con una función a secas. ¿Qué tal si creamos un decorador que sea también una clase?

Como sabes en Python todo es o puede ser un objeto. Una función, una clase pueden ser objetos también si así lo deseamos. Podemos utilizar una clase como decorador haciéndola (invocable). Para ello debemos utilizar la función **__call__** que nos permitirá emular un objeto como si fuese una función.

Así que venga sin más vueltas vamos a crear una clase que llamaremos “midecorador” de la siguiente manera:

```
1.  #!/usr/bin/env python
2.  # -*- coding: utf-8 -*-
3.  #
4.  #  decoradores.py
5.  class midecorador(object):
6.      def __init__(self, func): #Damos como parámetro una función
7.          print ("He construido la clase")
8.          func() #Llamamos a la función
9.
10.     def __call__(self): #La definimos como llamable
11.         print ("Soy una clase llamada mediante call")
12.
13.
14.
15. def hablar():
16.     print ("Hola soy la función hablar")
17.
18. matias = midecorador(hablar) #instanciamos y llamamos la función hablar brindandola
como argumento
```

Resultado:

He construido la clase

Hola soy la función hablar

Instanciamos el objeto matias pasándole como argumento la función que queremos decorar con dicha clase. Y dicha función, recordemos que en este caso es invocada dentro del __init__. Por lo que se ejecutará automáticamente con el constructor de la clase siempre que se cree una instancia de esta clase.

Aunque también podemos recurrir al “@” para decorar sin instanciar.

```
1.  #!/usr/bin/env python
2.  # -*- coding: utf-8 -*-
3.  #
4.  #  decoradores.py
5.  class midecorador(object):
6.      def __init__(self, func): #Damos como parámetro una función
7.          print ("He construido la clase")
8.          func() #Llamamos a la función
9.
10.     def __call__(self): #La definimos como llamable
11.         print ("Soy una clase llamada mediante call")
12.
13.
14. @midecorador
15. def hablar():
16.     print ("Hola soy la función hablar")
```

Resultado:

He construido la clase

Hola soy la función hablar

Vemos que ni siquiera llamamos a la función, sino que solo con decorarla al definirla esta es llamada automáticamente.

Pero si nosotros queremos que la misma no sea llamada automáticamente **debemos almacenarla en el constructor y llamarla en el __call__**.

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
#
#  decoradores.py
class midecorador(object):
    def __init__(self, func): #Damos como parámetro una función
        print ("""He construido la clase""")
        self.func = func #La almacenamos en el constructor


    def __call__(self): #La definimos como llamable
        print ("Soy una clase llamada mediante call")
        self.func() #Ejecutamos la funcion en call


@midecorador
def hablar():
    print ("Hola soy la función hablar")

hablar() #Llamamos la función decorada
```

Resultado:

He construido la clase

Soy una clase llamada mediante call

Hola soy la función hablar

También podemos pasarle argumentos a esta función utilizando *args y **kargs en el __call__.

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
#
#  decoradores.py
class midecorador(object):
    def __init__(self, func): #Damos como parámetro una función
        print ("""He construido la clase""")
        self.func = func #La almacenamos en el constructor


    def __call__(self, *args, **kargs): #La definimos como llamable
        print ("Soy una clase llamada mediante call")
        self.func(*args, **kargs) #Ejecutamos la funcion en call


@midecorador
def hablar(mensaje):
    print (mensaje)

hablar("Soy un argumento para el parámetro mensaje") #Llamamos la función decorada
```

Resultado:

He construido la clase

Soy una clase llamada mediante call

Soy un argumento para el parámetro mensaje

De esta forma hemos aprendido a crear nuestros propios decoradores que nos permitirán añadir funcionalidad extra a una función.