UTN.BA | Centro de e-Learning
Centro de e-Learning SCEU UTN - BA. Medrano 951 2do piso
(1179) // Tel. +54 11 7078- 8073 / Fax +54 11 4032 0148
www.sceu.frba.utn.edu.ar/e-learning

Diplomatura en Python:

Curso de Python nivel avanzado

UTN.BA | Centro de e-Learning
Universidad Tecnológica Nacional
Facultad Regional Buenos Aires
Secretaría de Cultura y Extensión Universitaria

Centro de e-Learning SCEU UTN - BA. Medrano 951 2do piso
(1179) // Tel. +54 11 7078- 8073 / Fax +54 11 4032 0148
www.sceu.frba.utn.edu.ar/e-learning

Módulo 1:

Nuevas estructuras

Unidad 3:

Decoradores

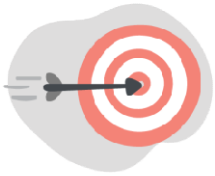
Centro de e-Learning SCEU UTN - BA. Medrano 951 2do piso
(1179) // Tel. +54 11 7078- 8073 / Fax +54 11 4032 0148
www.sceu.frba.utn.edu.ar/e-learning

Presentación

En el transcurso de esta unidad aprenderemos a crear y trabajar con decoradores, un decorador es un objeto que puede tomar una función, método o clase y extender su funcionalidad, insertando código que se ejecuta automáticamente al final de la declaración del elemento sobre el cual actúa. De hecho los decoradores mismos toman la forma de objetos que pueden a su vez ser llamados por otros decoradores.

La decoración de clases se puede utilizar desde las versiones de python 2.6 y 3.0.

Centro de e-Learning SCEU UTN - BA. Medrano 951 2do piso
(1179) // Tel. +54 11 7078- 8073 / Fax +54 11 4032 0148
www.sceu.frba.utn.edu.ar/e-learning



Objetivos

Que los participantes logren…

Aprendan a crear decoradores.

Comprendan como pueden aplicarse más de un decoradora sobre una función, método o clase.

Puedan crear y utilizar decoradores con parámetros.

Centro de e-Learning SCEU UTN - BA. Medrano 951 2do piso
(1179) // Tel. +54 11 7078- 8073 / Fax +54 11 4032 0148
www.sceu.frba.utn.edu.ar/e-learning

Bloques temáticos

1.- Introducción – Decoradores.

2.- Decoradores de Clases.

3.- Anidamientos.

4.- Argumentos.

5.- Opciones de retención de información.

Centro de e-Learning SCEU UTN - BA. Medrano 951 2do piso
(1179) // Tel. +54 11 7078- 8073 / Fax +54 11 4032 0148
www.sceu.frba.utn.edu.ar/e-learning

1.- Usando PyMongo

1. Introducción - Decoradores.

Los decoradores nos permiten adicionar lógica a nuestros programas, la cual se ejecuta automáticamente cuando se llama a una función o se ejecuta la instancia de una clase.

Un decorador es una forma de especificar la administración o incremento de código en funciones o clases. Se pueden tomar como objetos que procesan otros objetos. Tenemos dos tipos:

Primero: Decoradores de funciones agregados desde python 2.4 que permiten redefinir una función en tiempo de definición de la misma.

Segundo: Decoradores de clases agregado desde python 2.6 que permiten redefinir una clase en tiempo de definición de la misma.

Implementación

Podemos comenzar por considerar en la práctica a un decorador como una función que envuelve a otra que estamos llamando, la toma como entrada, opera con la misma y nos retorna un resultado que puede ser la misma función o no. La sintaxis básica es la siguiente, en donde el decorador se debe especificar antes de la función llamándolo por su nombre y anteponiéndole el símbolo de @:

```
def decorator(mi_funcion):
    # Procesa la función "MiFuncion "
    return (MiFuncion u otro objeto)

@decorator
def mi_funcion(): ..
```

mi_funcion()

Nota: El decorador es aplicado en tiempo de ejecución de la función.

Un ejemplo nos aclarará un poco la idea, consideremos que tenemos una función destinada a sumar dos elementos que llamamos "Sumar" y que le aplicamos un decorador "**decoratorMultiplicarPor10**" cuya función es que tome el resultado que la función retornaría y lo multiplique por 10, como se muestra a continuación:

decorador1.py

```
1   def decorator_multiplicar_por10(funcion):
2       def envoltura(*args):
3           print(funcion(*args) * 10)
4       return envoltura
5
6   @decorator_multiplicar_por10
7   def agregar(a, b):
8       c = a + b
9       return c
10
11  agregar(3, 5)
```

Al ejecutarlo obtenemos:

```
80
```

Si no hubiéramos aplicado el decorador, el resultado debería haber sido 8. Este simple ejemplo nos demuestra cómo podemos agregar una lógica extra a la función.

El operador es llamado cuando se lee la línea que invoca al decorador y su ejecución se realizar cuando se llama a la función sobre la cual se aplica. En realidad cuando se llama a la función, se ejecuta la función "Envoltura" que retorna el decorador, la cual toma la función y sus argumentos y puede realizar alguna operación.

Nota: Atención, si el decorador se utiliza sobre un método de clase, el primer argumento args[0] es una instancia de la clase por lo que esta construcción si puede utilizarse sobre métodos de clase.

Técnicamente, esta versión de función anidada funciona porque Python crea un objeto de método enlazado y, por lo tanto, pasa la instancia de clase de sujeto al argumento self solo cuando un atributo de método hace referencia a una función simple; Cuando en su lugar hace referencia a una instancia de una clase, la instancia de la clase llamante se pasa a self para dar a la clase llamable acceso a su propia información de estado.

En lugar de utilizar una función envolvente, podemos utilizar una clase con la ayuda del método __call__, y utilizar atributos de instancia, como se muestra a continuación:

decorador2.py

```
1   class decoratorMultiplicarPor10:
2       def __init__(self, func):
3           self.func = func
4       def __call__(self, *args):
5           print(self.func(*args) * 10)
6
7   @decoratorMultiplicarPor10
8   def agregar(a, b):
9       c = a + b
10      return c
11
12  agregar(3, 5)
```

Al ejecutarlo obtenemos:

```
80
```

OJO: **NO SIRVE PARA INTERCEPTAR MÉTODOS DE CLASE, SOLO FUNCIONES**.
El problema con esto es que self en el método __call__ recibe una instancia de "decorador" pero *args no incluye una instancia de la clase en la cual quiero utilizar el decorador sobre un método.

Ejemplo

Veamos un ejemplo más interesante antes de pasar al siguiente punto, consideremos un decorador destinado a contar las veces que se ha llamado a una determinada función y presentar sus parámetros utilizando el operador de instancia __call__.

decorador3.py

```
1   class ContarLlamadas:
2       def __init__(self, funcion):
3           self.numero_de_llamada = 0
4           self.funcion = funcion
5       def __call__(self, *args, **kwargs):
6           self.numero_de_llamada += 1
7           texto = '########## NUEVA LLAMADA A FUNCIÓN: '+
    self.funcion.__name__ +' ##########'
8           print(texto ,end='!\n')
9           print('Llamada número %s a la función %s' % (self.numero_de_llamada,
    self.funcion.__name__))
10          self.funcion(*args, **kwargs)
11          print('----Instancia de de corador contador----')
12          print(self)
13          print('----Argumentos de función----')
14          if args:
15              for i in args:
16                  print(i)
17
18          if kwargs:
19              for clave, valor in kwargs.items():
20                  print( "%s == %s" %(clave, valor))
21
22  @ContarLlamadas
23  def suma(a, b, c):
24      print(a + b + c)
25
26
27  @ContarLlamadas
28  def exponencial(x, y):
29      print(x ** y)
30
31  suma(1,2,3)
32  suma(a=4, b=5, c=6)
33  exponencial(2,2)
34  exponencial(2,6)
```

Centro de e-Learning SCEU UTN - BA. Medrano 951 2do piso
(1179) // Tel. +54 11 7078- 8073 / Fax +54 11 4032 0148
www.sceu.frba.utn.edu.ar/e-learning

Al ejecutarlo obtenemos:

```
######### NUEVA LLAMADA A FUNCIÓN: Suma #########!
Llamada número 1 a la función Suma
6
----Instancia de de corador contador----
<__main__.contarLlamadas object at 0x015031B0>
----Argumentos de función----
1
2
3
######### NUEVA LLAMADA A FUNCIÓN: Suma #########!
Llamada número 2 a la función Suma
15
----Instancia de de corador contador----
<__main__.contarLlamadas object at 0x015031B0>
----Argumentos de función----
a == 4
b == 5
c == 6
########## NUEVA LLAMADA A FUNCIÓN: Exponencial #########!
Llamada número 1 a la función Exponencial
4
----Instancia de de corador contador----
<__main__.contarLlamadas object at 0x01503230>
----Argumentos de función----
2
2
########## NUEVA LLAMADA A FUNCIÓN: Exponencial #########!
Llamada número 2 a la función Exponencial
64
----Instancia de de corador contador----
<__main__.contarLlamadas object at 0x01503230>
----Argumentos de función----
2
6
```

Nota: Cada función a la que se le aplica el decorador crea una nueva instancia de la función decorador con su propio contador de llamadas.

El código anterior funciona tanto en python2 como en python3, pero como comentamos solo se puede aplicar a funciones y no a métodos de instancia de clase. En el decorador self hace referencia a la instancia del decorador y no a la instancia de de la clase a cuyo método le aplicaríamos el decorador y la instancia de la clase no se encuentra incluida tampoco en *args.

2.- Decoradores de Clases

Los decoradores de funciones demostraron ser tan útiles que el modelo se extendió para permitir la decoración de clase a partir de Python 2.6 y 3.0. Inicialmente existió resistencia a su implementación debido a la superposición de roles con las metaclases; al final, sin embargo, fueron adoptados porque proporcionan una forma más sencilla de lograr muchos de los mismos objetivos. Los decoradores de clase están fuertemente relacionados con los decoradores de funciones; de hecho, utilizan la misma sintaxis, y patrones de codificación muy similares. Sin embargo, en lugar de envolver funciones o métodos individuales, los decoradores de clase son una forma de administrar clases o de terminar las llamadas de construcción de instancias con lógica adicional que administra o aumenta las instancias creadas a partir de una clase.

decorador4.py

```
1   def decorator(cls):
2       class Envoltura:
3
4           def __init__(self, *args):
5               self.instanciaDeClaseOriginal = cls(*args) #genero instancia de la clase
6   original
            def __getattr__(self, nombre):
7               print(self, ' -----  Instancia de Envoltura')
8               print(self.instanciaDeClaseOriginal, ' -----  Instancia de Auto.')
9               print('Nombre de atributos de clase Auto: ', nombre)
10              return getattr(self.instanciaDeClaseOriginal, nombre) #Retorna el valor del
    atributo de la clase C llamado
11      return Envoltura
```

```
12
13  @decorator
14  class Auto:
15      def __init__(self, color, marca):
16          self.color = color
17          self.marca = marca
18  x = Auto('Rojo', 'Renault')
19  print(x.color)
20  print(x.marca)
```

Al ejecutarlo obtenemos:

```
<__main__.decorator.<locals>.Envoltura object at 0x035632D0>    -----  Instancia de Envoltura
<__main__.Auto object at 0x035632F0>   -----  Instancia de Auto.
Nombre de atributos de clase Auto:  color
Rojo
<__main__.decorator.<locals>.Envoltura object at 0x035632D0>    -----  Instancia de Envoltura
<__main__.Auto object at 0x035632F0>   -----  Instancia de Auto.
Nombre de atributos de clase Auto:  marca
Renault
```

En este caso el decorador recuerda el nombre de la clase dentro de otra clase que retiene el nombre original y crea una instancia de la clase original cuando es llamada.

```
self.instanciaDeClaseOriginal = cls(*args)
```

Cuando el atributo es atrapado en la instancia, es interceptado por __getattr__ y delegado a la instancia de la clase original.

Centro de e-Learning SCEU UTN - BA. Medrano 951 2do piso
(1179) // Tel. +54 11 7078- 8073 / Fax +54 11 4032 0148
www.sceu.frba.utn.edu.ar/e-learning

3.- Anidamientos

Podemos realizar anidamientos de decoradores, tanto de funciones como de clases, anidar decoradores en funciones de la siguiente forma:

Estructura de anidamiento en funciones

```
@A
@B
@C
def f(...):
    ...
```

Esto es equivalente a considerar el siguiente orden:

```
def f(...):
    ...

f = A(B(C(f)))
```

Estructura de anidamiento en clases

```
@a
@b
@c
class C:
    ...

X = C()
```

Esto es equivalente a considerar el siguiente orden:

```
class C:
    ...

C= a(b(c(C)))
X = C()
```

Centro de e-Learning SCEU UTN - BA. Medrano 951 2do piso
(1179) // Tel. +54 11 7078- 8073 / Fax +54 11 4032 0148
www.sceu.frba.utn.edu.ar/e-learning

Veamos un ejemplo en el cual controlamos el estado de un motor, el primer decorador "**avisoCambioEstado**" la podemos destinar para agregarle una lógica de envío de cambio de estado a un dispositivo móvil, mientras que el segundo decorador lo podemos destinar para agregar una lógica extra en función del estado del motor.

decorador5.py

```
1   def cambio_estado(f):
2       def inner(*args, **kwargs):
3           if args[0]:
4               f(*args, **kwargs)
5               print ("El motor se ha encendido")
6           else:
7               f(*args, **kwargs)
8               print ("El motor se ha apagado")
9       return inner
10
11  def aviso_cambio_estado(f):
12      def inner(*args, **kwargs):
13          print ("Se envia un mensaje a dispositivo movil.")
14          f(*args, **kwargs)
15          print ("Se ejecuto %s" % f.__name__)
16      return inner
17
18  @cambio_estado
19  @aviso_cambio_estado
20  def estadoMotor(estado):
21      print (estado)
22
23  estado_motor(True)
24  print ("---"*20)
25  estado_motor(False)
```

Al ejecutarlo obtenemos:

```
Se envia un mensaje a dispositivo movil.
True
Se ejecuto estadoMotor
```

```
El motor se ha encendido
------------------------------------------------------------
Se envia un mensaje a dispositivo movil.
False
Se ejecuto estadoMotor
El motor se ha apagado
```

4.- Argumentos

Tanto los decoradores de funciones como los de clase también pueden tomar argumentos, aunque en realidad estos argumentos se transfieren a un llamador que, en efecto, devuelve el decorador, que a su vez devuelve un llamable. Por naturaleza, esto generalmente establece múltiples niveles de retención de estado. El siguiente es un esquema de la lógica utilizada:

```
def decorator(A, B):
    # Utiliza  A, B
    def actual_decorator(F):
        ## Este es como el contenido del decorador que trabajamos hasta aquí.
    return actual_decorator

@decorator(A, B)
def mi_funcion(x, y, z):
    ...
F(12, 23, 34)
```

Código equivalente:

```
def mi_funcion(x, y, z):
    ...
F = decorator(A, B)(F)
F(12, 23, 34)
```

Veamos un ejemplo en el cual creamos un decorador destinado a determinar si vamos a trabajar en modo de desarrollo o de producción, el parámetro “debug” de la función decoradora determina el modo de trabajo y dentro del decorador agregamos otra función que reciba la función sobre la cual aplicamos el decorador y sus argumentos.

Si debug es igual a “True”, retorna “Estoy en desarrollo”, mientras que si es “False” retorna “Estoy en producción”.

Centro de e-Learning SCEU UTN - BA. Medrano 951 2do piso
(1179) // Tel. +54 11 7078- 8073 / Fax +54 11 4032 0148
www.sceu.frba.utn.edu.ar/e-learning

decorador6.py

```
1   def modo_de_trabajo(debug=False):
2       def _modo_de_trabajo(funcion):
3           def interna(*args, **kwargs):
4               if debug:
5                   print ("Estoy en desarrollo")
6               else:
7                   print ("Estoy en producción")
8               for id, argumento in enumerate(args):
9                   print ("argumento %d:%s" % (id, argumento))
10              funcion(*args, **kwargs)
11          return interna
12      return _modo_de_trabajo
13
14  @modo_de_trabajo(False)
15  def registro_usuario(nombre, apellido):
16      print ("Registro de:  %s" % nombre)
17
18  registro_usuario("Juan", "Perez")
```

Al ejecutarlo obtenemos:

```
Estoy en producción
argumento 0:Juan
argumento 1:Perez
Registro de:  Juan
```

Centro de e-Learning SCEU UTN - BA. Medrano 951 2do piso
(1179) // Tel. +54 11 7078- 8073 / Fax +54 11 4032 0148
www.sceu.frba.utn.edu.ar/e-learning

5.- Opciones de retención de información.

Los decoradores de funciones tienen una variedad de opciones para retener la información de estado proporcionada en el momento de la decoración, para usar durante la llamada de función real. Por lo general, necesitan soportar varios objetos decorados y múltiples llamadas, pero hay varias maneras de implementar estas metas:

1) Los atributos de instancia.

2) Variables globales.

3) Variables NonLocal.

4) Atributos de función.

El caso de atributos de instancia ya lo hemos analizado cuando realizamos el decorador "**contarLlamadas**" en el ejercicio "**decorador3.py**", veamos una variante del mismo utilizando las otras opciones.

Variables globales.

Aquí el problema es que al utilizar variables globales el valor cambia con cada llamada de decorador por lo que el resultado obtenido no es el esperado!!!

decorador7.py

```
1   numero_de_llamada = 0
2   def contarLlamadas(funcion):
3       def envoltura(*args, **kwargs):
4           global numero_de_llamada
5           numero_de_llamada += 1
6           print('Llamada número %s a la función %s' % (numero_de_llamada,
    funcion.__name__))
7           return funcion(*args, **kwargs)
8       return envoltura
9
10  @contar_llamadas
11  def suma(a, b, c):
12      print(a + b + c)
```

```
13
14  @contar_llamadas
15  def exponencial(x, y):
16      print(x ** y)
17
18  suma(1,2,3)
19  suma(a=4, b=5, c=6)
20  exponencial(2,2)
21  exponencial(2,6)
```

Al ejecutarlo obtenemos:

```
Llamada número 1 a la función Suma
6
Llamada número 2 a la función Suma
15
Llamada número 3 a la función Exponencial
4
Llamada número 4 a la función Exponencial
64
```

Variables NonLocal.

Utilizar nonlocal es una buena alternativa en python 3.x, aunque no está disponible en python 2.x. La variante nos quedaría así:

decorador8.py

```
1   def contar_llamadas(funcion):
2       numero_de_llamada = 0
3       def envoltura(*args, **kwargs):
4           nonlocal numero_de_llamada
5           numero_de_llamada += 1
6           print('Llamada número %s a la función %s' % (numero_de_llamada,
    funcion.__name__))
7           return funcion(*args, **kwargs)
8       return envoltura
9
10  @contar_llamadas
```

Centro de e-Learning SCEU UTN - BA. Medrano 951 2do piso
(1179) // Tel. +54 11 7078- 8073 / Fax +54 11 4032 0148
www.sceu.frba.utn.edu.ar/e-learning

```
11  def suma(a, b, c):
12      print(a + b + c)
13
14  @contar_llamadas
15  def exponencial(x, y):
16      print(x ** y)
17
18  suma(1,2,3)
19  suma(a=4, b=5, c=6)
20  exponencial(2,2)
21  exponencial(2,6)
```

Al ejecutarlo obtenemos:

```
Llamada número 1 a la función Suma
6
Llamada número 2 a la función Suma
15
Llamada número 1 a la función Exponencial
4
Llamada número 2 a la función Exponencial
64
```

Atributos de función.

Finalmente, si deseamos que nuestro código funcione tanto en python 2.x como en 3.x podemos utilizar desde la versión 2.1 de python atributos de función de la siguiente forma:

```
función.atributo = valor
```

El código nos quearía así:

decorador9.py

```
1  def contar_llamadas(funcion):
2      def envoltura(*args, **kwargs):
3          envoltura.numero_de_llamada += 1
4          print('Llamada número %s a la función %s' % (envoltura.numero_de_llamada,
       funcion.__name__))
5          return funcion(*args, **kwargs)
6      envoltura.numero_de_llamada = 0
7      return envoltura
8
```

Centro de e-Learning SCEU UTN - BA. Medrano 951 2do piso
(1179) // Tel. +54 11 7078- 8073 / Fax +54 11 4032 0148
www.sceu.frba.utn.edu.ar/e-learning

```
9    @contar_llamadas
10   def suma(a, b, c):
11       print(a + b + c)
12
13   @contar_llamadas
14   def exponencial(x, y):
15       print(x ** y)
16
17   suma(1,2,3)
18   suma(a=4, b=5, c=6)
19   exponencial(2,2)
20   exponencial(2,6)
```

Al ejecutarlo obtenemos:

```
Llamada número 1 a la función Suma
6
Llamada número 2 a la función Suma
15
Llamada número 1 a la función Exponencial
4
Llamada número 2 a la función Exponencial
64
```

Centro de e-Learning SCEU UTN - BA. Medrano 951 2do piso
(1179) // Tel. +54 11 7078- 8073 / Fax +54 11 4032 0148
www.sceu.frba.utn.edu.ar/e-learning

Bibliografía utilizada y sugerida

Libros y otros manuscritos:

Programming Python 5th Edition – Mark Lutz – O'Reilly 2013

Programming Python 4th Edition – Mark Lutz – O'Reilly 2011

Manual online

https://docs.python.org/3.7/tutorial/

https://docs.python.org/3.7/library/index.html

https://docs.python.org/3/reference/datamodel.html