Seminario 14

* **Decoradores en Python.**
* **Usos para verificar tipos.**

# Equipo 5:

* Luis Ernesto Ibarra
* Luis Enrique Dalmau
* Laura Tamayo Blanco
* Alberto Helquera
* Jessy Gigato

¿Qué es el patrón Decorador (Decorator Pattern)?

Es un patrón que responde a la necesidad de añadir dinámicamente funcionalidades a Objetos.

Aplicabilidad:

* Añadir responsabilidades a objetos individuales de forma dinámica y transparente
* Responsabilidades de un objeto pueden ser retiradas
* Hay una necesidad de extender la funcionalidad de una clase, pero no hay razones para extenderlo a través de la herencia.
* Existe la necesidad de extender dinámicamente la funcionalidad de un objeto y quizás quitar la funcionalidad extendida.

¿Qué es un decorador en Python?

Un decorador en Python es una función que toma como primer parámetro una función y devuelve una función. También se pueden utilizar clases.

Sintaxis:

@Expr\_1

@Expr\_2

…

@Expr\_n

definition # function or class definition

Todas las expresiones Expr\_i deben implementar \_\_call\_\_ con posibilidad de recibir el parámetro de la función a decorar.

La sintaxis anterior solamente es azúcar sintáctica para:

definition = Expr\_1(Expr\_2(…(Expr\_n(definition))…)

Ejemplos:

En este ejemplo se puede observar cómo definir un decorador simple. En el llamado de la función decorada el código ejecutado es el de inner\_func en vez del target\_func.

def decorator(func): # Wrap Function

def inner\_func(\*args,\*\*kwargs): # Inner Function

# Something to do before the func call

value = func(\*args,\*\*kwargs)

# Something to do after the func call

return value

return inner\_func

@decorator

def target\_function(\*args,\*\*kwargs):

# Function body

pass

target\_function() # Function call

En este ejemplo se puede observar cómo definir un decorador que tome argumentos. Veamos a que se traduce esto para ver con más claridad lo que sucede:

target\_function = param\_decorator(1,2,3)(target\_function)

Así se puede apreciar que la función devuelta en la primera llamada tomará como argumento target\_function retornando a inner\_func como target\_function.

def param\_decorator(param0,param1,param2): # Params Function

def wrap\_func(func): # Wrap Function

def inner\_func(\*args,\*\*kwargs): # Inner Function

# Something to do before the func call

value = func(\*args,\*\*kwargs)

# Something to do after the func call

return value

return inner\_func

return wrap\_func

@param\_decorator(1,2,3)

def target\_function(\*args,\*\*kwargs):

# Function body

pass

target\_function()

En este ejemplo se puede observar cómo definir un decorador con una clase. Veamos a que se traduce esto para ver con más claridad lo que sucede:

target\_function = Decorator(target\_function)

Así se puede apreciar que la instancia de la clase devuelta se almacena en target\_function, y cada vez que se llame a esta función (instancia de clase), llamará a \_\_call\_\_, haciendo parecer que se esta llamando a una función.

class Decorator:

def \_\_init\_\_(self,func): # Wrap Function

self.func = func

def \_\_call\_\_(self, \*args,\*\*kwargs): # Inner Function

# Something to do before the func call

value = self.func(\*args,\*\*kwargs)

# Something to do after the func call

@Decorator

def target\_function(\*args,\*\*kwargs):

# Function body

pass

target\_function()

En este ejemplo se puede observar cómo definir un decorador con una clase que tome argumentos. Veamos a que se traduce esto para ver con más claridad lo que sucede:

target\_function = ArgumentDecorator(1,2,3)(target\_function)

Así se puede apreciar que la instancia de la clase devuelta en la primera llamada, llamará a \_\_call\_\_ tomando como argumento target\_function retornando a inner\_func como target\_function.

class ArgumentDecorator:

def \_\_init\_\_(self,\*args): # Params Function

self.args = args

def \_\_call\_\_(self, func): # Wrap Function

def inner\_func(\*args,\*\*kwargs): # Inner Function

# Something to do before the func call

value = func(\*args,\*\*kwargs)

# Something to do after the func call

return inner\_func

@ArgumentDecorator(1,2,3)

def target\_function(\*args,\*\*kwargs):

# Function body

pass

target\_function()

Análisis general de los ejemplos

Como se pudo apreciar en los ejemplos anteriores, se dividió las partes que componen un decorador, para mayor entendimiento, en tres:

* Inner Function: Es la función que se ejecutara cada vez que se llame a la función decorada.
* Wrap Function: Es la función encargada de capturar la función a decorar. Debe de tomar solamente un argumento, el de la función, y de devolver una Inner Function.
* Params Functions: Son aquellas que se utilizan con el operador @ las cuales pueden tomar parámetros para personalizar el comportamiento de la función decorada. Viene acompañada de una Wrap Function y una Inner Function.

Ejercicio 1:

1. Implemente un decorador **typeCheck** que reciba los tipos de los parámetros de la función a decorar y veriﬁque en cada invocación que los objetos pasados como argumentos correspondan con dichos tipos.

En Python las variables y los parámetros de las funciones no tienen tipo estático, pero los objetos dentro de estas sí tienen tipos. Por lo tanto se hizo un decorador que en tiempo de ejecución, verificara que los parámetros anotados (Ej: a:int) de la función se correspondieran con su tipo de entrada. Si se anota el tipo de salida, esta también es verificada.

Vea el código en Code/decorator.py

Ejercicio 3:

3. ¿Tendría sentido modificar el lenguaje para tener variables y parámetros con tipos? ¿La definición del lenguaje obliga a la necesidad de variables y parámetros sin tipo?

Como todos sabemos en Python las variables y parámetros son dinámicos por la forma en que se construyó el lenguaje, esto se debe a como en runtime el intérprete se encarga de buscar en los diccionarios de los objetos las definiciones de sus métodos o atributos. Con ello se logra una mayor flexibilidad, libertad y refactibilidad del código creado, pero ocurre también que se produce una cierta inseguridad en el código debido a que no se puede saber hasta ejecutar el código si este es correcto y además se pierde en rendimiento debido al preprocesamiento que debe hacer el intérprete para poder ejecutar el programa.

Por ello muchos se preguntarán si tendría sentido modificar el lenguaje para tener variables y parámetros con tipo, pero antes de responder a esta interrogante ojeemos primero lo que dice el Zen de Python:

* Bello es mejor que feo.
* Explícito es mejor que implícito.
* Simple es mejor que complejo.
* Complejo es mejor que complicado.
* Plano es mejor que anidado.
* Disperso es mejor que denso.
* La legibilidad cuenta.
* Los casos especiales no son tan especiales como para quebrantar las reglas.
* Lo práctico gana a lo puro.
* Los errores nunca deberían dejarse pasar silenciosamente.
* A menos que hayan sido silenciados explícitamente.
* Frente a la ambigüedad, rechaza la tentación de adivinar.
* Debería haber una -y preferiblemente sólo una- manera obvia de hacerlo.
* Aunque esa manera puede no ser obvia al principio a menos que usted sea holandés.
* Ahora es mejor que nunca.
* Aunque nunca es a menudo mejor que ya mismo.
* Si la implementación es difícil de explicar, es una mala idea.
* Si la implementación es fácil de explicar, puede que sea una buena idea.
* Los espacios de nombres (namespaces) son una gran idea ¡Hagamos más de esas cosas!

Tim Peters, El Zen de Python

Se aprecia que algunos de los puntos que más se resaltan son la simplicidad, practicidad, facilidad de explicar y belleza del código, esto nos hace pensar que el dinamismo de tipado para los creadores de Python es considerado una muy buena idea y hace que nos cuestionemos de si realmente el tipado tiene lugar en un lenguaje así.

Veamos algunos de los puntos fuertes del tipado estático:

* Detección temprana de errores de programación (normalmente en tiempo de compilación).
* Un mayor rendimiento (por la posibilidad de compilador de hacer mayor optimización al código).
* Posibilidad de implementar mecanismos como el Intelisense.
* Mayor robustez del código.
* Facilita la documentación del código escrito.

En el caso de agregar tipado en Python, al no haber proceso de compilación, no se puede lograr el primer punto, ni el segundo. El tercero no tendría mucho sentido ya que como el lenguaje es predominantemente dinámicamente tipado este no funcionaría en la mayoría de los casos por lo que ciertamente se pierden varias de las ventajas que viene con el tipado estático al usar tipos en Python.

Pero, aun así, es cierto que escribir código en Python con tipos le ofrecería una mayor robustez al código escrito, facilitaría la documentación del mismo y haría más fácil el trabajo con las IDEs, por lo cual en grandes módulos en que sea necesario testear el código este podría ser bastante útil.

Por tanto, consideramos que el uso de tipos en Python tendría sentido; aunque siempre intentando no abusar de él y perder las ventajas que tiene el propio lenguaje, así como la ideología con que este fue concebido.