PONTIFICIA UNIVERSIDAD CATÓLICA DE CHILE
ESCUELA DE INGENIERÍA
DEPARTAMENTO DE CIENCIA DE LA COMPUTACIÓN

IIC2233 - Programación Avanzada
2° **semestre 2015**

Actividad 08

Decoradores

Instrucciones

En lenguajes como Java o C# existe un tipo de polimorfismo llamado overloading, en el cual se sobredefine un método en una clase para que dependiendo del tipo de parámetros recibidos al ejecutar se realice cierta acción específica.

A continuación se observa un ejemplo en el cual si se llama el método `Sumador.sumar` con dos `int` se obtendrá un distinto comportamiento si es que se llama con dos `str`.

```
class Sumador
{
        static void sumar(string s1, string s2)
        {
                Console.WriteLine(s1 + " " + s2 + "!");
        }

        static void sumar(int n1, int n2)
        {
                Console.WriteLine("El resultado es: {0}", n1 + n2);
        }
}
```

Para algunos programadores resulta complejo pasar de lenguajes tipados como estos a lenguajes con mayores libertades como Python. Es por esto que usted, como un buen programador, ha decidido crear decoradores para ayudar a los pobres que no conocen el maravilloso mundo de Python. Para esto usted creará dos decoradores

1. Decorador para forzar tipado

En primer lugar deberá crear un decorador que le permita tipar los argumentos de una función, es decir, forzar que una función solo tome cierto tipo de argumentos. En el caso de recibir otro tipo de argumentos, simplemente imprime un mensaje de error.

```
@tipar(int, int)
def sumar(a, b):
    # Esta funcion solo acepta sumar dos <int>
    return a + b
```

2. Decorador para hacer Overloading

En segundo lugar, deberá programar un decorador que le permita hacer overloading de métodos en una clase. Para esto tendrá que completar el código de la clase `Overload` en el archivo entregado. Esto le permitirá que el siguiente código emule el código original en C#

```
class Sumador:
    @Overload
    def sumar(self):
        # Esta funcion se debe llamar cuando no se ingresa ningun argumento
        pass

    @sumar.overload(str, str)
    def sumar(self, s1, s2):
        # Esto se se ejecuta al llamar sumar() con dos <str> como parametros
        print(a + " " + b "!")

    @sumar.overload(int, int)
    def sumar(self, n1, n2):
        # Esto se se ejecuta al llamar sumar() con dos <int> como parametros
        print("El resultado es:", n1 + n2)
```

Para esto usted deberá completar los siguientes métodos: (Vea los tips para un ejemplo de una clase decorador)

- `__init__()`
  Recibirá como argumento una función e inicializará el decorador
- `overload()`
  Recibirá como argumento los tipos de parámetros que aceptará la siguiente función. Este método debe retornar un decorador que tome una función y la asocie a los tipos recién nombrados y la guarde de alguna forma en la clase.
- `__call__()`
  Recibirá como argumento una cantidad arbitraria de parámetros. Dependiendo de los tipos de estos parámetros se deberá ejecutar la función correcta, previamente guardada por el método `overload`.

Como verá, se encuentra definido el método `__get__`. No se preocupe por este método. Se encuentra definido para que la clase pueda ser ejecutada correctamente como decorador, usted no debe hacerle ningún cambio.

Requerimientos

- Completar el decorador `tipar` para poder tipar las funciones.
- Completar el código la clase `Overload` para que cumpla la funcionalidad requerida. Para esto es necesario como mínimo rellenar las funciones `__init__`, `__call__` y `overload`.

To - DO

- (5.00 pts) Decorador `tipar`
- (3.00 pts) Decorador `Overload`

Tips

- No se enrede con el código del `"__main__"`. Está ahí simplemente para que usted pueda probar su actividad. Una vez terminados los decoradores, el programa debiese correr sin lanzar errores.
- Puedes verificar si un objeto es de un tipo especifico mediante el método `isinstance(obj, type)` donde retornará `True` sí y solo sí el objeto `obj` es de tipo `type`
- Para utilizar clases como decordores simplemente tienes que hacer que el `__init__` tome como parámetro la función a decorar e implementar el método `__call__` de la clase. Esto te permite implementar más métodos que en una función decoradora para generar comportamientos interesantes.
  Por ejemplo, acá se ve el código de un decorador que permite guardar varias funciones mediante el método `agregar`. Cuando se ejecuta a una función decorada por esta clase, se llama al método `__call__`, y en consecuencia, se ejecutan todas las funciones que han sido guardadas.

```
class MuchasFunciones:
    def __init__(self, func):
        self._funcs = [func]

    def agregar(self, func):
        self._funcs.append(func)
        # es importante que este metodo retorne una referencia a self
        return self

    def __call__(self, *args, **kwargs):
        return [func(*args, **kwargs) for func in self._funcs]

    def __get__(self, obj, cls):
        def caller(*args, **kwargs):
            return self(obj, *args, **kwargs)
        return caller
```

Este decorador permite ejecutar varias funciones a la vez y retorna todos los valores en una lista. Se utilizaría de la siguiente forma:

```
class Math:
    @MuchasFunciones
    def operacion(self, a, b):
        # suma
        return a + b

    @operacion.agregar
    def operacion(self, a, b):
        # multiplicacion
        return a * b
m = Math()
suma, mult = m.operacion(2, 4) # [6, 8]
```