Ingeniería de Software: wrappers y decoradores en Python

En el mundo de la ingeniería de sistemas, Python es uno de los lenguajes más temidos y respetados. El respeto, no es algo que deba perderse, pero el miedo, debería ser erradicado. Y eso, es lo que pretendo lograr con esta serie de dos entregas: otorgar una explicación clara, simple y definitiva sobre los 4 "monstruos" del lenguaje. En la entrega final, wrappers y decoradores.

En la edición anterior de The Original Hacker, se hizo una explicación profunda y detallada sobre las funciones *lambda* y los ***closures*** dejando pendientes los ***wrappers*** y ***decoradores*** para esta entrega. Y de poner a los *wrappers* y decoradores en blanco sobre negro, es que nos encargaremos ahora.

Antes de comenzar es necesario entender que:

1. No se puede hablar de *wrappers* sin haber entendido previamente qué son los *closures*. Por lo tanto, si no lo haz hecho aún, te recomiendo leer las **páginas 5 a 8** de **The Original Hacker Nº3**, edición que puedes obtener en http://library.originalhacker.org/biblioteca/revista/ver/19

2. No se puede hablar de decoradores sin hablar de wrappers, puesto que básicamente ambos términos hacen referencia a lo mismo, pero con una sutil diferencia a nivel teórico-conceptual.

Aclarado esto, estamos en condiciones de continuar con lo nuestro.

Lo primero que debemos recordar, es que **un *closure* es una función que dentro de ella contiene a otra función** la cual es retornada cuando el *closure* es invocado:

```
def closure(parametro1):
    def funcion(parametro2):
        print parametro1 + parametro2
    return funcion

foo = closure(10)  # foo ahora es la función interna del closure
print foo(200)     # Imprime:  210
```

```
print foo(500)     # Imprime:  510
```

No olvides repasar el artículo sobre closures publicado en The Original Hacker Nº3: http://library.originalhacker.org/biblioteca/revista/ver/19

Un decorador (*decorator*), **es aquel *closure* que como parámetro recibe a una función (llamada función "decorada") como único argumento**:

```
def decorador(funcion_decorada):

    def funcion():
        pass

    return funcion
```

Mientras que **un *wrapper*** no es más que la **función interna de un *closure* que a la vez sea de tipo decorador** (función a la que en los ejemplos anteriores llamamos "funcion" a secas):

```
def decorador(funcion_decorada):

    def wrapper():
        pass

    return wrapper
```

La **función decorada** deberá ser invocada por el *wrapper*:

```
def decorador(funcion_decorada):

    def wrapper():
        return funcion_decorada()

    return wrapper
```

El decorador no se invoca como una función normal. Éste es llamado con una sintaxis especial:

```
@decorador
```

La sintaxis anterior se debe colocar en la línea anterior a la definición de la función decorada. De esta forma, el nombre de la función decorada es pasado como parámetro de forma automática sin tener que invocar nada más:

```
@decorador
def funcion_decorada():
    print 'Soy una función decorada'
```

Hasta aquí entonces, tenemos que:

- Un ***closure*** es una función que dentro de ella define otra función.
- Un **decorador** es un *closure* que recibe una función como parámetro.
- Una **función decorada** es la que se pasa como parámetro a un decorador.
- Un ***wrapper*** es la función interna del decorador, encargada de retornar a la función decorada.

¿Cómo funcionan y se accionan los wrappers y decoradores?

Cuando una función es decorada, el decorador se acciona de forma automática en el momento que el *script* es ejecutado:

```
#!/usr/bin/env python


def decorador(funcion):
    print 'Soy el decorador()'

    def wrapper():
        print 'Soy el wrapper()'
        return funcion()

    return wrapper


@decorador
def funcion_decorada():
    print 'Soy la funcion_decorada()'
```

Al ejecutar este *script* SIN haber invocado ninguna función, podemos ver como el decorador ya ha actuado:

```
eugenia@cococha-gnucita:~/borrador$ ./foo.py
Soy el decorador()
eugenia@cococha-gnucita:~/borrador$
```

Cuando la función decorada es invocada, el decorador ya la habrá reemplazado por el *wrapper*, retornando a éste en lugar de la función original, es decir, retorna al *warpper* en lugar la función decorada, tal como muestra el siguiente ejemplo:

```
#!/usr/bin/env python
```

```
def decorador(funcion):
    print 'Soy el decorador()'

    def wrapper():
        print 'Soy el wrapper()'
        # return funcion()

    return wrapper


@decorador
def funcion_decorada():
    print 'Soy la funcion_decorada()'


funcion_decorada()
```

El resultado de la ejecución del *script* será el siguiente:

```
eugenia@cococha-gnucita:~/borrador$ ./foo.py
Soy el decorador()
Soy el wrapper()
eugenia@cococha-gnucita:~/borrador$
```

Como bien puede verse, al haber invocado a `funcion_decorada()`, no fue ésta quien se ejecutó sino que ha sido la función `wrapper()`.

Pero, tal y como se dijo al comienzo, el *wrapper* será quien tras su ejecución, invoque a la función decorada:

```
#!/usr/bin/env python


def decorador(funcion):
    print 'Soy el decorador()'

    def wrapper():
        print 'Soy el wrapper()'
        return funcion()

    return wrapper


@decorador
def funcion_decorada():
    print 'Soy la funcion_decorada()'


funcion_decorada()
```

El resultado ahora, será el siguiente:

```
eugenia@cococha-gnucita:~/borrador$ ./foo.py
Soy el decorador()
Soy el wrapper()
Soy la funcion_decorada()
eugenia@cococha-gnucita:~/borrador$
```

El **orden de ejecución en *wrappers* y decoradores** se puede establecer como el siguiente:

1. **decorador** (automáticamente al ejecutar el *script* reemplazando la función decorada por el *wrapper*)
2. ***wrapper*** (al invocar a la función decorada)
3. **función decorada** (luego de ejecutarse el *wrapper*)

> *Puede decirse entonces que conceptualmente,* ***un decorador es un closure que se encarga de reemplazar a la función decorada por su función interna, a la cuál se denomina wrapper***

¿Para qué usaríamos un wrapper?

En primer lugar, notarás que la pregunta NO ha sido para qué usar un decorador, sino para que usar un *wrapper*. Esto es debido a que la respuesta a para qué utilizar un decorador es fácilmente deducible: para implementar un *wrapper*.

Los *wrappers* o **envolturas**, suelen utilizarse cuando se tienen funciones que ANTES de ejecutar su verdadera funcionalidad, realizan acciones redundantes. Un ejemplo muy típico, es cuando se utilizan bloques `try` y `except`:

```
def get_template():
    try:
        with open('template.html', 'r') as archivo:
            return archivo.read()
    except:
        return 'ERROR INTERNO'


def calcular(partes=0, total=100):
    try:
        return total / partes
    except:
        return 'ERROR INTERNO'
```

Como podemos ver, tenemos dos funciones que utilizan bloques try y except (generalmente, se tienen muchísimas más). Una forma de manejar los errores en estas funciones, sería utilizar un wrapper:

```
def intentar(funcion):
```

```
    # Usamos *args y **kwargs ya que tenemos una función que necesita argumentos
    def wrapper(*args, **kwargs):
        try:
            return funcion(*args, **kwargs)
        except:
            return 'ERROR INTERNO'

    return wrapper
```

De esta forma, solo deberíamos decorar a las funciones que anteriormente utilizaban los bloques `try` y `except`:

```
@intentar
def get_template():
    with open('template.html', 'r') as archivo:
        return archivo.read()


@intentar
def calcular(partes=0, total=100):
    return total / partes
```

Si invocáramos a `calcular` con los argumentos por defecto, se produciría un error al intentar dividir por cero, pero sería manejado por el *wrapper*:

```
eugenia@cococha-gnucita:~/borrador$ ./bar.py
ERROR INTERNO
eugenia@cococha-gnucita:~/borrador$
```

Frecuentemente, lo mismo que sucede con el **manejo de errores** en cuanto a acciones repetidas en distintas funciones, suele darse con acciones como el **control de acceso** (usado por ejemplo, para verificar si un usuario tiene permisos suficientes antes de efectuar una determinada acción) o el **registro de actividades** (conocido como *logging*).

Un uso menos frecuente aunque paradójicamente podríamos mencionarlo como imprescindible, es para el **control de datos** mediante **filtros de saneamiento** o de **tratamiento**, como se muestra en el siguiente ejemplo Web para Python sobre Apache con WSGI:

```
def get_post_data(funcion):  # decorador

    def wrapper(environ):  # claramente este es el wrapper del decorador
        _POST = {}

        try:
            datos = environ['wsgi.input'].read().split('&')
            for par in datos:
                key, value = par.split('=')
```

```
                _POST[key] = unquote(value).replace('+', ' ')
                key, value = (None, None)
        except:
            pass

        return funcion(_POST)  # retorna a la función decorada una vez termina lo anterior

    return wrapper


@get_post_data      # llamada al decorador
def guardar(POST):  # función decorada
    nombre = POST['nombre']
    apellido = POST['apellido']
    # ...


@get_post_data         # llamada al decorador
def actualizar(POST):  # función decorada
    pass


@get_post_data       # llamada al decorador
def eliminar(POST):  # función decorada
    pass
```

Como puede verse en el ejemplo anterior, la llamada al decorador es una sintaxis abreviada equivalente a:

```
get_post_data(guardar)
def guardar(POST):
    nombre = POST['nombre']
    apellido = POST['apellido']
    # ...


get_post_data(actualizar)
def actualizar(POST):
    pass


get_post_data(eliminar)
def eliminar(POST):
    pass
```

Sin embargo, la nueva sintaxis propuesta por la incorporación de decoradores, hace la lectura del código fuente mucho más simple.