Ingeniería de Software: De lambdas, closures, wrappers y decoradores. Desmitificando Python - Parte I

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 de este mes, lambdas y closures.

Llevo años poniendo caras de sorpresa y asombro cada vez que escucho a mis colegas hablar sobre **funciones lambda**, ***closures***, ***wrappers*** y **decoradores** en Python. Y llevo la misma cantidad de tiempo tratando de dar a mis alumnos un mensaje "tranquilizador" cuando me consultan alarmados por estos "cuatro monstruos".

No se qué es lo que rodea a Python que lo hace estar involucrado en explicaciones complejas que pretenden hacerlo parecer un lenguaje místico, parte de un culto religioso o logia secreta. Pero lo que sí se, es que **Python no es un lenguaje místico ni complejo**. Es tan solo un lenguaje de programación más. Potente, sí. Agradable y prolijo, también. Pero sigue siendo un lenguaje más, que por más complejidad que se le quiera atribuir, siempre habrá oportunidad de desmitificarla y encontrar una explicación sencilla. Y **a eso he venido con este artículo; a desmitificar un hito en la historia de este maravilloso lenguaje.**

Funciones lambda

Muchas veces -y muy a mi pesar- llamadas "funciones anónimas", las funciones lambda en Python, no son más que **una forma de definir una función** común y corriente, de una única instrucción de código, **en una única línea**.

Es decir, una función lambda es la forma de definir una función que tradicionalmente podría escribirse de forma común, en una sola línea de código. Pero esto, **solo podrá hacerse con aquellas funciones cuyo algoritmo, no posea más de una instrucción**.

Definir una función lambda

La siguiente función:

```
def mifuncion(nombre):
    return "Hola %s!" % nombre
```

Con lambda, podría definirse en una sola línea de código, ya que posee una única instrucción. Para ello, se utilizaría la siguiente instrucción:

```
mifuncion = lambda nombre: "Hola %s!" % nombre
```

Visto de esta forma, hasta se corre el riesgo de perderle el respeto a las funciones lambda. De hecho lo anterior, casi no requiere de una explicación. Sin embargo, la daré por si aún quedan dudas.

> *Una función **lambda** es una forma de **definir una función** de una sola instrucción, **en una sola línea de código***

Una función lambda es una forma de definir una función de una sola instrucción, en una sola línea de código, claro que, con una sintaxis particular:

```
variable = lambda parametro1, parametro2, parametro9: instrucción a ser retornada
```

Al igual que las funciones comunes, **las funciones lambda admiten parámetros por omisión** y la principal diferencia es que **el resultado de toda instrucción siempre es retornado**:

```
neto = lambda bruto, iva=21: bruto + (bruto * iva / 100)
```

Invocar funciones lambda

La llamada a las funciones lambda es idéntica a la de funciones comunes. Es decir, el hecho de ser funciones asignadas a una variable, no significará que éstas se ejecutarán sin ser invocadas. Por el contrario, son funciones que al igual que las funciones comunes, deberán ser invocadas a través de la variable a la cual se las asignado:

```
neto = lambda bruto, iva=21: bruto + (bruto * iva / 100)

print neto(100)
```

```
# Retorna 121

print neto(200)
# Retorna 242

print neto(bruto=100, iva=19)
# Retorna 119
```

El tipo de datos de una variable asignada como función lambda

Una variable a la cual se le asignado la definición de una función lambda, muy intuitivamente, **es una variable de tipo "función lambda"**. Y esta obviedad no es en demérito de Python; muy por el contrario, **es lo que convierte a Python en un lenguaje con coherencia lógica**:

```
>>> neto = lambda bruto, iva=21: bruto + (bruto * iva / 100)
>>> neto
<function <lambda> at 0xb6a7e0d4>
```

Closures

Los *closures* son funciones que dentro de ellas, definen otra función. Hasta aquí, es una definición que poco dice sobre los closures, pero si a lo anterior agregamos que al ser invocado un closure, retorna la función que define dentro, la cosa comienza a verse un poco mejor.

> *Un **closure** es entonces, una **función que define otra función y la retorna** al ser invocado.*

Por ejemplo:

```
def closure():

    def funcion_interna():
        return 1

    return funcion_interna
```

Al llamar a `closure()` lo que en realidad obtendríamos como resultado, sería otra función, la función llamada `funcion_interna()`:

```
variable = closure()  # Ahora variable es una función, la función funcion_interna()
print variable()      # Imprimirá 1 (1 es el retorno de funcion_interna()
```

Visto de esta forma, intentamos únicamente introducir el concepto sintáctico y funcional de un *closure*. Ahora nos toca ver cómo y para qué se utilizan.
Sabemos entonces, que se trata de funciones que definen otra función dentro y la retornan pero también, debemos agregar que **dichas funciones internas, tienen la capacidad de reconocer y recordar el valor de variables y parámetros definidos dentro del *closure***:

```
def closure(parametro):

    def funcion():
        return parametro + 1  # parametro es de la función closure()

    return funcion

variable = closure(parametro=1)
print variable()  # Imprime 2
```

Y hasta aquí, nuevamente, lo máximo que hemos hecho es entender cómo funciona un *closure* y cómo se define e invoca, pero ninguno de los ejemplos sería válido en términos de buenas prácticas, pues ninguno justificaría su implementación.

> *Un closure DEBE tener una razón de ser. De hecho, mi consejo es evitarlos toda vez que sea posible resolver el planteo sin utilizarlos. Pues dificultan la lectura del código y su entendimiento, cuando en realidad, deberían estar allí para facilitarlo.*

Un ejemplo tal vez más esclarecedor de un *closure* bien implementado, podría ser el siguiente:

```
def calcular_iva(alicuota):

    def estimar_neto(importe_bruto):
        return importe_bruto + (importe_bruto * alicuota / 100)

    return estimar_neto


# Productos gravados con el 21%
get_neto_base_21 = calcular_iva(21)
harina = get_neto_base_21(10)
arroz = get_neto_base_21(8.75)
azucar = get_neto_base_21(12.5)

# Productos gravados con el 10.5%
get_neto_base_105 = calcular_iva(10.5)
tv = get_neto_base_105(12700)
automovil = get_neto_base_105(73250)
```

El requerimiento anterior, podría haber requerido definir dos funciones independientes (`get_neto_base_21` y `get_neto_base_105`) con prácticamente el mismo algoritmo y una importante redundancia.

En su defecto, podría haberse resuelto mediante una única función con 2 parámetros, la cual hubiese requerido replicar el valor de uno de los parámetros incansablemente, en las reiteradas llamadas.

Pues allí, es donde tenemos la **justificación para implementar un *closure*:**

- evitar definir múltiples funciones con algoritmos redundantes
- evitar la sobrecarga de parámetros

> *Se justificará el uso de **closures** cuando su finalidad consista en **evitar la redundancia y la sobrecarga de parámetros***

Pero para entender mejor el móvil, veamos en código qué oportunidades teníamos:

```
# Recurrir a algoritmos redundantes

def get_neto_base_21(importe_bruto):
    return importe_bruto + (importe_bruto * 21 / 100)    # línea redundante


def get_neto_base_105(importe_bruto):
    return importe_bruto + (importe_bruto * 10.5 / 100)  # línea redundante


harina = get_neto_base_21(10)
arroz = get_neto_base_21(8.75)
azucar = get_neto_base_21(12.5)
tv = get_neto_base_105(12700)
automovil = get_neto_base_105(73250)


# Recurrir a la sobrecarga de parámetros

def get_neto(alicuota_iva, importe_bruto):
    return importe_bruto + (importe_bruto * alicuota_iva / 100)
```

Ahora, en las siguientes llamadas, notar el valor recurrente en el primer parámetro las primeras 3 veces y las dos últimas:

```
harina = get_neto(21, 10)
arroz = get_neto(21, 8.75)
azucar = get_neto(21, 12.5)

tv = get_neto(10.5, 12700)
automovil = get_neto(10.5, 73250)
```

Justificar la implementación de un closure

Como puede verse en los ejemplos anteriores, ninguna de las alternativas "huele bien". Si se recurre a la redundancia, por cada tipo de gravamen, deberíamos tener una función. Por ese motivo, la alternativa queda descartada por completo.

En cambio, si se recurre a la sobrecarga de parámetros, se podría optar por definir un parámetro por omisión. Sin embargo ¿cuál sería el valor de ese parámetro si se tuviesen 1000 productos gravados con un 21% y 1000 con el 10.5%?

Definitivamente, se trataría de decisiones tomadas de forma arbitraria y no, basadas en deducciones lógicas.

Es entonces, que el parámetro recurrente, motiva el *closure* y el algoritmo redundante, a la función interna.

Para concluir esta primera entrega, quiero cerrar con uno de esos trucos que generalmente nadie desvela:

> ***La mejor implementación de un closure*** *no será la que haya sido perfectamente planificada sobre un papel en blanco, sino* ***aquella alcanzada tras un cuidadoso refactoring***.

En la próxima entrega, nos concentraremos en los *wrappers* y decoradores; dos formas sintácticamente similares a los *closures* pero conceptualmente diferentes.