Seminario 18 Python V

Equipo 9

- Adrian Tubal Páez Ruiz
- Olivia González Peña
- Juan Carlos Casteleiro Wong
- Gabriela Mijenes Carrera
- Reinaldo Barrera Travieso

Implemente un módulo llamado `functionTools` donde se encuentren las siguientes definiciones:

- `fixParams`: Función que permite "fijar" valores como argumentos de funciones. Por ejemplo, fijar el valor 2 como primer argumento de una función `f` con tres parámetros consiste en obtener una función `g` de 2 parámetros de manera que `g(a, b)` sea equivalente a hacer `f(2, a, b)` . Para esto `fixParams` recibe como primer argumento la función seguida de los valores que se quieren fijar. Por ejemplo, la siguiente línea fija los valores 6 y 7 como segundo y cuarto argumento respectivamente de una función de cuatro parámetros: `g = fixParams(f, _, 6, _, 7)`. El valor especial _ (guión bajo) debe ser definido en el módulo functionTools para indicar que un parámetro no tiene valor fijo. Luego de la línea del ejemplo hacer `g(1, 2)` es equivalente a hacer `f(1, 6, 2, 7)`.
- `_`: valor especial para indicar un parámetro sin valor fijo en la función `fixParams`.
- `rearrangeParams`: Función que permite cambiar el orden de los parámetros de una función. Recibe una función de n parámetros y una permutación de 0, ..., n − 1 y devuelve una función con los parámetros ordenados según dicha permutación. El valor en la posición i de la permutación indica la posición del parámetro i de la función original en la función resultante. Veamos los siguientes ejemplos:
  - `g = rearrangeParams(sum, 3, 2, 1, 0)`: la función `g` es equivalente a la función `sum` con los parámetros invertidos, o sea, `sum(a, b, c, d)` es equivalente a hacer `g(d, c, b, a)`.
  - `g = rearrangeParams(sum, 0, 1, 3, 2)` : la función `g` es equivalente a la función `sum` con los dos últimos parámetros invertidos, o sea, `sum(a, b, c, d)` es equivalente a hacer `g(a, b, d, c)`.

`fixParams`

Definimos el tipo _ para utilizarlo posteriormente en la función `fixParams` como indicador de parámetro libre.

```
class Ignore:
    pass


_ = Ignore()
```

```
def fixParams(func, *args, **kwargs):
    fixed_args = args
    fixed_kwargs = kwargs

    def newFunction(*args, **kwargs):
        i = j = 0
        newArgs = []
        while i < len(fixed_args) and j < len(args):
            if type(fixed_args[i]) is not Ignore:
                newArgs.append(fixed_args[i])
                i += 1
            else:
                newArgs.append(args[j])
                j += 1
        while i < len(fixed_args):
            if type(fixed_args[i]) is not Ignore:
                newArgs.append(fixed_args[i])
            i += 1
        while j < len(args):
            newArgs.append(args[j])
            j += 1

        for key, value in fixed_kwargs.items():
            kwargs[key] = value
        func(*newArgs, **kwargs)

    return newFunction
```

En el código anterior vamos a identificar y describir las siguientes variables:

- `func`: es la función a la cual se le quiere fijar valores en los argumentos.
- `fixed_args`: es la tupla con el conjunto de valores fijos y libres. Por ejemplo `(_, 6, _, 7)`.
- `fixed_kwargs`: es el conjunto de parámetros nombrados, si existen, serán los últimos argumentos de la función. ¿Para qué queremos usar `kwargs` en este caso? Si `func` tiene varios `kwargs` sería cómodo poder fijarle valores a un conjunto de esos `kwargs`, esto nos permite cambiar el valor por defecto que tenía.
- `newFunction`: es la función con los cambios realizados y que será devuelta por `fixParams`.
- `newArgs`: es el nuevo vector de argumentos que será utilizado.

En el cuerpo de `newFunction` se pueden ver tres ciclos `while` seguidos, que van a implementar la estrategia de reemplazo de los argumentos. El reemplazo de argumentos sigue la siguiente estrategia:

- El i-ésimo elemento de `args` será puesto en el i-ésimo argumento libre (`_`) de `fixed_args`. Es decir si `args = (1,3)` y `fixed_args = (0,_,2,_)` el resultado es `(0,1,2,3)`.
- Si hay más elementos en `args` que espacios libres en `fixed_args`, entonces simplemente el resto se agrega al final, por ejemplo, `args = (1,3,4,5)` y `fixed_args = (0,_,2,_)` el resultado es `(0,1,2,3,4,5)`.
- Si hay más espacios libres en `fixed_args` que elementos en `args`, entonces son ignorados los que sobran, por ejemplo, `args = (1,3)` y `fixed_args = (0,_,2,_,_,_)` el resultado es `(0,1,2,3)`

Tener en cuenta que la definición de la función `newFunction` está dentro de la función `fixParams`, esto es posible en Python ya que las funciones son ciudadanos de primera clase.

Se dice que en un lenguaje las funciones son ciudadanos de primera clase cuando se pueden tratar como cualquier otro valor del lenguaje, es decir, cuando se pueden almacenar en variables, pasar como parámetros o devolver como valor de retorno de una función, sin ningún tratamiento especial.

Notar también que `newFunction` termina con el llamado a la función original `func` con los argumentos modificados.

`rearrangeParam`

```
def rearrangeParam(f, *p):
    def retf(*arg):
        narg = [0 for i in range(len(p))]
        for i in range(len(p)):
            narg[p[i]] = arg[i]
        arg = narg
        return f(*arg)
    return retf

def f(x, y):
    return x - y

g = rearrangeParam(f, 1, 0)
print(g(1, 2))              # 1
```

La función `rearrangeParam` se encarga de recibir una función `f`, una lista de parámetros `p` y devolver la función `retf`, que ya tiene los parámetros reconfigurados.

Dentro del cuerpo de `retf` encontramos a `narg`, una lista donde se van a guardar los parámetros ya reconfigurados según la permutación `p` recibida en `rearrangeParam`. Una vez que tengamos la nueva configuración devolvemos la función con la nueva lista de parámetros.

Usando las ventajas de Python podemos dar una implementación más reducida, como se muestra a continuación:

```
def rearrangeParam(f, *args):
    def newFunc(*kargs):
        return f(*(kargs[i] for i in args))
    return newFunc
```

En este caso `newFunc` retorna la función `f` con una tupla, que tiene los parámetros ya reconfigurados, ahorrándonos la declaración de una variable para almacenar la configuración deseada.

Ejemplos de prueba:

```
def sum(*args):
    s = 0
    for x in args:
        s += x
    return x

g = rearrangeParam(sum, 3, 2, 1, 0)
print(g(3, 2, 1, 0) == sum(0, 1, 2, 3))     #True
```

```
def minus(x,y):
    return x - y

g = rearrangeParam(minus, 1, 0)
print(g(1, 2) == minus(2, 1))               #True
```

Name Binding (Enlace de Nombres)

Name Binding es la asociación entre un nombre y un objeto (valor). En Python hacemos este vínculo a través del operador de asignación (=).

```
a = 1
b = 2
```

En este ejemplo se crean los nombres `a`, `b` y se le vinculan los valores 1, 2, respectivamente. Si quisiéramos hacer una asignación al estilo:

```
c = a
```

En Python se crea un nombre `c` asociado al objeto 1, el cual ya está vinculado al nombre `a`. No se copia el objeto 1, ni se crea uno nuevo, solo se introduce un nuevo nombre para este objeto; entonces, ambos, `a` y `c` están vinculados al objeto 1. Por tanto tendríamos tres nombres y dos objetos:

```
a ---> 1 <--- c
b ---> 2
```

Si luego de esto hiciéramos:

```
a = b
```

Ahora el nombre `a` está vinculado al objeto 2, sin tener implicaciones sobre `c`, que se mantiene vinculado al objeto 1.

```
       1 <--- c
b ---> 2 <--- a
```

Binding (Enlace) está estrechamente relacionado con scoping (alcance, ámbito), ya que el scope determina qué nombres se vinculan a qué objetos dependiendo de la ubicación del código del programa. En Python la resolución de nombres se rige por la llamada regla LEGB (Local, Enclosing, Global, Built-In):

- `Local`: es el cuerpo de cualquier función o expresión lambda. Este ámbito contiene los nombres que define dentro de la función, que serán visibles solo desde el código de la función. Se crea en la llamada a la función, no en la definición, por lo que tendrá tantos ámbitos locales diferentes como llamadas. Esto es cierto incluso si llama a la misma función varias veces o de forma recursiva; cada llamada dará como resultado la creación de un nuevo ámbito local.
- `Enclosing (Nonlocal)`: solo existe para funciones anidadas. Si el ámbito local es una función interna o anidada (inner) entonces el ámbito envolvente (enclosing scope) es el de la función externa; este ámbito contiene tanto los nombres definidos en su cuerpo como los definidos en el enclosing scope.
- `Global`: es el alcance más alto en un programa, script o módulo de Python. Los nombres en este ámbito son visibles desde cualquier parte de su código.
- `Built-In`: este ámbito contiene nombres como palabras claves, funciones, excepciones y otros atributos integrados en Python. Están disponible en cualquier parte de su código y se carga automáticamente cuando se ejecuta un programa o script.

La regla LEGB es un tipo de procedimiento de búsqueda de nombres, que determina el orden en que Python busca los nombres. Se busca secuencialmente en el ámbito local, envolvente (enclosing), global, built-in; si el nombre existe se obtendrá la primera aparición, de lo contrario se obtendrá un error.

En Python podemos especificar a quién referenciar con las palabras claves `global` y `nonlocal`. Por ejemplo:

```
def f():
    global value1
    value1 = 100
    value2 = 200

value1 = 0
value2 = 0
f()
print(value1)    # 100
print(value2)    # 0
```

Fue posible modificar el valor de la variable `value1` ya que fue referenciada a través de la palabra clave `global`.

En caso de tener funciones anidadas podemos lograr un comportamiento similar haciendo uso de la palabra clave `nonlocal` teniendo en cuenta las siguientes restricciones:

- `nonlocal` debe hacer referencia a una variable existente en el cuerpo de alguna función que la envuelve.
- no puede acceder a las variables globales.

Sin utilizar `nonlocal`:

```
x = 0
def outer():
    x = 1
    def inner():
        x = 2
        print("inner:", x)

    inner()
    print("outer:", x)

outer()
print("global:", x)
```

Output:

```
inner: 2
outer: 1
global: 0
```

Usando `nonlocal`:

```
x = 0
def outer():
    x = 1
    def inner():
        nonlocal x
        x = 2
        print("inner:", x)

    inner()
    print("outer:", x)

outer()
print("global:", x)
```

Output:

```
inner: 2
outer: 2
global: 0
```

Usando `global`:

```
x = 0
def outer():
    x = 1
    def inner():
        global x
        x = 2
        print("inner:", x)

    inner()
    print("outer:", x)

outer()
print("global:", x)
```

Output:

```
inner: 2
outer: 1
global: 2
```

Currying

El término currying se refiere al proceso de tomar una función que recibe n argumentos y convertirla en n funciones de un argumento cada una. LLeva el nombre de Haskell Curry, matemático y lógico del siglo XX, cuyo trabajo sentó las bases para los lenguajes de programación funcionales.

Veamos el siguiente código:

```
def Suma(a, b, c, d):
    return a + b + c + d
```

Esta función recibe cuatro parámetros para resolverse y devuelve la suma de ellos al terminar. Cuando aplicamos currying a `Suma` lo que obtenemos es una función que toma un argumento y retorna otra función que recibe el siguiente, este proceso se repite hasta haber obtenido todos los argumentos, será entonces que se devuelva la suma de ellos.

A continuación proponemos una implemetación de un decorador para aplicar currying

```
def currying(func):
    def curried(*args, **kwargs):
        if len(args) + len(kwargs) >=
        func.__code__.co_argcount:
```

```
            return func(*args, **kwargs)
        def newFunc(*args2, **kwargs2):
            return curried(*(args + args2))
        return newFunc
    return curried

@currying
def Suma(a, b, c, d):
    return a + b + c + d
```

```
print(Suma(1)(2)(3)(4))     # 10
```

Aplicación parcial de funciones

Si una función f espera n parámetros y es llamada con m (m < n) normalmente el compilador nos dice que necesitamos más argumentos para f. En el ejemplo anterior si llamáramos a `Suma` con solo dos argumentos sucedería esto. Aplicar parcialmente una función es precisamente lo que el compilador nos diría que es un error: aplicamos una función a un número menor de parámetros que los que espera, y obtenemos una función que recibe el resto de los parámetros que hacen falta y devuelve un valor del tipo que retorna la función original.

```
from functools import partial

def Suma(a, b, c, d):
    return a + b + c + d

Suma7 = partial(Suma, b = 5, c = 2)
print(Suma7(a = 1, d = 3))  # 11
```

La función `Suma7` es una aplicación parcial de `Suma` con los parámetros `b` y `c` fijados con los valores 5 y 2, respectivamente.

Diferencias entre currying y aplicación parcial de funciones

Es fácil confundir estos dos conceptos: ambos toman una función que recibe n parámetros y retornan una que espera un número menor de parámetros, la diferencia está en que en aplicación parcial de funciones se fijan con valores específicos algunos parámetros mientras que currying toma una función y proporciona una nueva función que acepta un solo argumento y devuelve la función especificada con su primer argumento establecido (esto nos permite representar funciones con múltiples argumentos como una serie de funciones de argumento único).

Curiosidad

En Haskell todas las funciones reciben exactamente un parámetro. Cuando queremos crear una que reciba múltiples parámetros hacemos uso de currying creando una serie de funciones donde cada una reciba un parámetro.