Programación Avanzada
Daniel Gregorio Longino
Tarea 5

1. Funciones lamda

Además de la sentencia `def`, Python también proporciona una forma de expresión que genera objetos de función. Por su similitud con una herramienta del lenguaje Lisp, se llama `lambda`. Al igual que `def`, esta expresión crea una función a la que se llamará más tarde, pero devuelve la función en lugar de asignarle un nombre. Esta es la razón por la cual las funciones lambda a veces se conocen como funciones anónimas (es decir, sin nombre).

Una función `lambda` tiene la siguiente sintaxis:

`lambda` *`argumento1,..., argumentoN : expresión usando los argumentos`*

Ejemplo de una función con la sentencia `def`:

```
>>> def function(x, y, z):
        return x+y+z

>>> function(2, 3, 4)
9
```

Se puede lograr el mismo efecto que en el ejemplo anterior con una expresión `lambda` de la siguiente manera:

```
>>> f = lambda x, y, z : x+y+z
>>> f(2, 3, 4)
9
```

2. Herramientas de la programación funcional

El Python actual combina el soporte para múltiples paradigmas de programación: procedimental (con sus declaraciones básicas), orientado a objetos (con sus clases) y funcional. Para el último de estos, Python incluye un conjunto de componentes integrados que se utilizan para la programación funcional: herramientas que aplican funciones a secuencias y otros iterables. Este conjunto incluye herramientas que invocan funciones en los elementos de un iterable (`map`); filtrar elementos según una función de prueba (`filter`); y aplicar funciones a pares de elementos y ejecutar resultados (`reduce`).

2.1. map

Una de las cosas más comunes que hacen los programas con listas y otras secuencias es aplicar una operación a cada elemento y recopilar los resultados: seleccionar columnas en las tablas de la base de datos, incrementar los campos de pago de los empleados en una empresa, analizar los archivos adjuntos de correo electrónico, etc. Python tiene varias herramientas que facilitan la codificación de tales operaciones. Por ejemplo, actualizar todos los elementos en una lista se puede hacer fácilmente con un bucle `for`:

```
>>> aux = [1, 2, 3, 4]
>>> actualizar = []
>>> for x in aux:
        actualizar.append(x+10)


>>> actualizar
[11, 12, 13, 14]
```

Pero debido a que esta es una operación tan común, Python también proporciona funciones integradas que hacen la mayor parte del trabajo por usted. La función `map` aplica una función a cada elemento en un objeto iterable y devuelve una lista que contiene todos los resultados de la llamada a la función. La función map() toma dos parámetros; una función que realiza alguna acción a cada elemento de un iterable, y un iterable tal como un conjunto, lista, tupla, etc.

Por ejemplo:

```
>>> def inc(x):
        return x+10

>>> list(map(inc, aux))
[11, 12, 13, 14]
```

Debido a que `map` espera que se pase y aplique una función, también resulta ser uno de los lugares donde comúnmente aparece `lambda`:

```
>>> list(map(lambda x: x+3, aux))
[4, 5, 6, 7]
```

Aquí, la función agrega 3 a cada elemento en la lista `aux`; como esta pequeña función no se necesita en ningún otro lugar, se escribió en línea como una lambda.

2.2. filter

La función de `map` es un representante principal y relativamente sencillo del conjunto de herramientas de programación funcional de Python. Sus parientes

cercanos, `filter` y `reduce`, seleccionan los elementos de un iterable en función de una función de prueba y aplican funciones a pares de elementos, respectivamente.

La función `filter`toma dos parámetros; una función y un iterable. Dado que este también devuelve un iterable, `filter` (como `range`) requiere una llamada a `list` para mostrar todos sus resultados. Por ejemplo, la siguiente llamada de `filter` selecciona los elementos en una secuencia que son mayores que cero:

```
>>> list(range(-5,5))
[-5, -4, -3, -2, -1, 0, 1, 2, 3, 4]
>>> list(filter((lambda x: x>0),range(-5,5)))
[1, 2, 3, 4]
```

2.3. reduce

La función `reduce`, que es una función integrada simple en Python 2.X pero que vive en el módulo functools en Python 3.X, es más compleja. Acepta un iterable para procesar, pero no es un iterable en sí mismo: devuelve un solo resultado. Esta función toma dos parámetros; una función y un iterable.

En el primer paso, se seleccionan los dos primeros elementos de la secuencia y se obtiene el resultado. El siguiente paso se vuelve a aplicar la misma función al resultado obtenido anteriormente y el número que sigue al segundo elemento y el resultado se almacena nuevamente. Este proceso continúa hasta que no quedan más elementos en el contenedor. El resultado final se devuelve y se imprime en la consola.

A continuación se presentan dos llamadas a la función `reduce` para obtener la suma y el producto de los elementos de una lista:

```
>>> from functools import reduce
>>> reduce((lambda x, y: x + y), [1, 2, 3, 4])
10
>>> reduce((lambda x, y: x * y), [1, 2, 3, 4])
24
```

3. Functools

`functools` es un módulo estándar que provee una serie de funciones que actúan sobre otras funciones. Más específicamente pueden aplicarse a cualquier objeto que implemente el método `__call__` . Todas las funciones del módulo son bastante diversas entre sí, compartiendo únicamente dicha particularidad: operan sobre otras funciones.

Entre ellas se encuentran `partial()`, para "congelar" una función con determinados argumentos; `lru_cache()`, para almacenar en una memoria caché

el resultado de una función; `singledispatch()`, que provee la posibilidad de implementar funciones genéricas; entre otras.

3.1. partial

Ésta es probablemente una de las funciones más últiles en toda la librería estándar.

`partial()` recibe una función A con sus respectivos argumentos y retorna una nueva función B que, al ser llamada, equivale a llamar a la función A con los argumentos provistos.

```
>>> from functools import partial
>>> def add(a, b):
        return a + b

>>> f = partial(add, 7, 5)
>>> f()
12
```

3.2. lru_cache

`functools.lru_cache()` (introducido en la versión 3.2) es un decorador para almacenar en una memoria caché el resultado de una función. Es decir, si una función es llamada 4 veces con los mismos argumentos, las últimas tres llamadas simplemente retornan un valor guardado en memoria, sin ejecutar realmente dicha función. Esto, en tareas pesadas, es una considerable optimización de memoria y velocidad.

```
from functools import lru_cache
from urllib.request import urlopen

def read_url(url):
    with urlopen(url) as r:
        return r.read()


print(len(read_url("https://www.recursospython.com")))
print(len(read_url("https://www.recursospython.com")))
print(len(read_url("https://www.recursospython.com")))
print(len(read_url("https://www.recursospython.com")))
```

3.3. cmp_to_key

Convierte una función de comparación en una función clave. La función de comparación debe devolver 1, -1 y 0 para diferentes condiciones. Se puede utilizar en funciones clave como `sorted()`, `min()`, `max()`.

```
from functools import cmp_to_key

def cmp_fun(a, b):
    if a[-1] > b[-1]:
        return 1
    elif a[-1] < b[-1]:
        return -1
    else:
        return 0

list1 = ['geeks', 'for', 'geeks']
l = sorted(list1, key = cmp_to_key(cmp_fun))
print('sorted list :', l)

>>>
=========== RESTART: C:/Users/52556/Downloads/borrar.py
sorted list : ['for', 'geeks', 'geeks']
>>>
```