Nano Taller de Python

Charla 5: “Programación Funcional”

Sergio Davis <sergdavis@gmail.com>

Royal Institute of Technology (KTH), Estocolmo, Suecia
Grupo de Nanomateriales (GNM), Santiago, Chile

13 de enero 2009, de 13:00 a 14:00

Parte I

Programación Funcional

Esquema de trabajo

En la sesión anterior vimos cómo usar clases, herencia e implementar polimorfismo en Python.
Ahora veremos cómo:

- Obtener listas a partir de expresiones
- Usar funciones anónimas (`lambda`)
- Definir generadores
- Usar expresiones generadoras para ahorrar memoria
- Aplicar técnicas funcionales a nuestros programas

Con esto cubrimos el último de los paradigmas de Python, lo cual nos permite copiar estilos de programación de otros lenguajes como `LISP` y `Haskell`

¿Qué es la programación funcional?

- En vez de diseñar el programa como instrucciones ejecutándose una a una, se diseña como definiciones abstractas de funciones, y aplicaciones de estas funciones para *mapear* un valor a otro
- Esto es una representación más matemática (similar a plantear *teoremas*) que mecánica de los procesos
- Python se inspiró para esto en lenguajes académicos como LISP y Haskell

Un ejemplo...

Compare:

```
b = False
for x in Listado('/home'):
    if PuedoLeer(x) and x == 'sdavis':
        b = True

print b
```

con:

```
v = [x for x in Listado('/home') if PuedoLeer(x)]
print 'sdavis' in v
```

¿Cuándo un lenguaje es funcional?

Para que merezca el apellido *funcional*, un lenguaje debería:

- Tratar a las funciones como objetos, manipulables como cualquier valor
- Esto es, crear funciones y pasarlas como argumento a otras funciones, que las podrán llamar, incluso definir funciones que retornen otras funciones como resultado
- Poder crear funciones anónimas, por ejemplo para usarlas como argumento de otra función
- Poder crearse clausuras, funciones parcialmente evaluadas pero con argumentos aún libres:

```
def suma(x, y): return x+y
def CreaClausura():
    def suma5(y): return suma(5, y)
    return suma5   # devuelve una funcion!

s = CreaClausura() # s es una funcion
print s(3)         # imprime 8
```

Funciones son objetos en Python

Una función en Python es un objeto, que puede ser pasado como argumento a otra función:

```
def Tabular(f, a, b, n):
    dx = (b-a)/float(n-1)
    return [f(a+i*dx) for i in range(n)]

from math import sin, pi
sintable = Tabular(sin, 0.0, 2.0*pi, 100)
```

Funciones son valores copiables

```
from math import pi, sin, cos
pimedio = 0.5*pi
f = sin
print f(pimedio)

# g es primero sin, luego cos
for g in [sin, cos]: print g(pimedio)

# intercambia las funciones sin y cos
# no lo intenten en casa...!
sin, cos = cos, sin
print "sin(pi/2) es ", sin(pimedio) # cos
print "cos(pi/2) es ", cos(pimedio) # sin
```

Mapear funciones con `map`

La función `map` aplica una función *f* a cada elemento de una secuencia y devuelve la lista de los resultados:

```
def f(x): return x**3

y = map(f, range(8))
print y
```

```
[0, 1, 8, 27, 64, 125, 216, 343]
```

`map` en el caso anterior es equivalente a:

```
y = [f(x) for x in range(8)]
```

Filtrar valores con `filter`

La función `filter` devuelve una secuencia conteniendo sólo los elementos que cumplen cierta condición:

```
def f(x): return (x > 5 and x < 10)
y = filter(f, range(20))
print y
```

```
[6, 7, 8, 9]
```

`filter` en el caso anterior es equivalente a:

```
y = [x for x in range(20) if f(x)]
```

Reducir listas con `reduce`

La función `reduce` transforma una secuencia de valores en un sólo valor, aplicando una función de dos valores de manera acumulativa:

```
def suma(x, y): return x+y

print reduce(suma, range(10))
# 0+1+2+3+...+9 = 45
```

Funciones anónimas (`lambda`)

Muchas veces una función es tan sencilla que no merece darle un nombre definiéndola con `def`. En este caso se puede usar una función anónima o `lambda`.
Por ejemplo, en vez de

```
def f(x): return x**3

y = map(f, range(8))
print y
```

Podemos escribir:

```
y = map(lambda x: x**3, range(8))
print y
```

Note que `lambda` no lleva `return`

Más sobre `lambda`

- `lambda` puede tomar cualquier número de argumentos:

```
lambda x, y, z: sqrt(x**2+y**2+z**2)
```

- Asignar el resultado de `lambda` a una variable es, para todos los efectos, equivalente a definirla con `def`:

```
mifuncion = lambda x, y: x**2+y**2
print mifuncion(5, 3)
```

```
def mifuncion(x, y): return x**2+y**2
print mifuncion(5, 3)
```

Compresión de listas

Proporciona una sintaxis mucho más clara que `map` y `filter`.

Sintaxis:

[*expresión(x)* `for x in` *secuencia* `if` *condición*]

Ejemplos:

```
y = [ z**2 for z in range(1, 6) ]
# [1, 4, 9, 16, 25]
y = [ '%.2f' % sqrt(x) for x in range(20)
      if x % 3 == 0 ]
# [ '0.00', '1.73', '2.45', '3.00', '3.46',
#   '3.87', '4.24' ]
```

```
lin = file('archivo.txt').readlines()
y = [L.partition('=') for L in lin if '=' in L]
```

Un ejemplo completo:
(Programación por “wishful thinking”)

Tarea: leer este archivo y transformarlo en un diccionario. También recuperar las claves en una lista aparte.

```
# estos son los parametros
# de mi programa
  foo = 5
   bar = 73
  alpha = 488
     beta = 0.0
  gamma = 123
```

Programación por “wishful thinking” consiste en imaginarse como nuestro programa principal se vería elegante... **y escribirlo tal cual!** (todo el “andamiaje” que se necesita para que funcione se escribirá después)

Es decir, el programa se comienza desde la idea más general, asumiendo que los detalles finos ya los programó alguien.

Un ejemplo completo (1):

(Programación por "wishful thinking")

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

# Primero escribimos el codigo tal
# y como quisieramos que se viera...
# mientras mas intuitivo mejor

def LeeArchivo(nombre):
    # info tiene inicialmente valores por defecto
    info = { 'alpha': 5, 'beta': 7, 'gamma': 3.45 }

    lineas = file(nombre).readlines()
    lineas = [x for x in lineas if not Ignorable(x)]
    separadas = [Separar(x) for x in lineas]
    claves = [AgregarA(info, x) for x in separadas]
    return (claves, info)
```

Un ejemplo completo (2):

(Programación por "wishful thinking")

```
# A continuacion definimos nuestro
# arsenal de funciones utilitarias
# que hacen que todo funcione como
# estaba planeado al principio

# True si la linea se puede ignorar
def Ignorable(x):
    return x.startswith('#') or x.strip() == ''

# Devuelve la particion pero 'stripeada'
def Separar(x):
    return [w.strip() for w in x.partition('=')]

# Agregar a diccionario
def AgregarA(diccio, part):
    diccio[part[0]] = part[2]
    return part[0]
```

Un ejemplo completo (3):
(Programación por “wishful thinking”)

Finalmente, llamamos al programa principal:

```
#
# Aqui llamamos al programa principal
# Podria ser mas obvio?
#
claves, info = LeeArchivo('archivo.txt')
print claves
print info
```

```
['foo', 'bar', 'alpha', 'beta', 'gamma']
{'alpha': '488', 'beta': '0.0', 'foo': '5',
'bar': '73', 'gamma': '123'}
```

Expresiones generadoras

Considere el siguiente caso de uso de comprensión de listas:

```
cubos = [x**3 for x in range(100000000)]
print [z for z in cubos if z % 1234 == 0]
```

`cubos` contiene los cubos de los primeros 100 millones de enteros. Como es una lista, se consume bastante memoria...
Es posible conseguir lo mismo usando una *expresión generadora* en vez de una compresión de listas:

```
cubos = (x**3 for x in xrange(100000000))
print [z for z in cubos if z % 1234 == 0]
```

Ahora `cubos` es una expresión generadora, no se guarda completa en memoria sino que los valores se generan a medida que son utilizados por la lista en `print`.

Expresiones generadoras

```
cubos = (x**3 for x in xrange(100000000))
print [z for z in cubos if z % 1234 == 0]
```

Sintaxis:

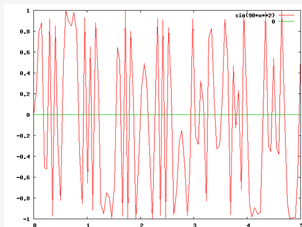
(*expresión(x)* `for x in` *secuencia* `if` *condición*)

Una expresión generadora **no es una lista**, sino otro tipo de secuencia, un **generador**. Es un iterable así que puede ir en un `for` explícito, en una comprensión de listas, o dentro de otra expresión generadora.

En el ejemplo anterior se reemplazó `range` por `xrange`. Esta segunda función es equivalente a `range`, pero no devuelve una lista de enteros... devuelve un generador de enteros. De haber usado `range` igual se guardarían los 100 millones de valores en memoria.

Ahora... a la práctica!

Problema:
Implementar un programa en Python que **estime** los cruces por cero de la función $f(x) = \sin(90 * x^2)$ entre $x = 0$ y $x = 5$.



Debería mostrar en pantalla la lista de los x_i para los cuales $f(x) \approx 0$
Hint: tabular la función bastante fino y chequear cuándo $f(x)$ cambia de signo entre un valor tabulado x_i y el anterior, x_{i-1}. Tomar el cero como el promedio entre x_i y x_{i-1}.
Por supuesto, todo esto se puede hacer a lo Fortran, pero ésta es una oportunidad para probar cosas funcionales, expresiones generadoras, al igual que el estilo “wishful thinking”.