Nano Taller de Python

Charla 3: “Contenedores: listas, tuplas, diccionarios”

Sergio Davis <sergdavis@gmail.com>

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

12 de enero 2009, de 15:00 a 16:00

Parte I

Contenedores: Tuplas, Listas, Diccionarios

Esquema de trabajo

En la sesión anterior vimos cómo programar en Python de manera "tradicional", al estilo BASIC, Fortran, Pascal o C
Ahora veremos cómo:

- Usar tuplas, listas y diccionarios
- Iterar sobre secuencias
- Trabajar con archivos de texto (*parsing*)

Con esto cubrimos el uso de Python como un procesador de texto, abandonando los scripts de `bash`, `awk` y `Perl`.

Idea básica de las secuencias

Una secuencia es una generalización del concepto de *arreglo*, una colección de valores **de tipos diversos**, que tienen distintos mecanismos de acceso a sus valores.

```
# Es posible recorrer una secuencia
# como un arreglo, por indice
A = [ 1, 3, 5, 7, 9, 11 ]
# i va desde 0 a 5
for i in range(len(A)):
    print A[i]
```

Iteración de secuencias

Además de poder acceder a los valores aleatoriamente, toda secuencia puede ser iterada, es decir, utilizada en un ciclo `for` para recorrer sus valores uno a uno.

```
# Tambien es posible iterar
# directamente sobre los elementos
# de una secuencia
A = [ 1, 3, 5, 7, 9, 11 ]
# x toma directamente
# los valores 1, 3, 5, 7, ...
for x in A:
    print x
```

Operaciones básicas sobre secuencias

`x in s`	`True` si *x* está en *s*
`x not in s`	`True` si *x* no está en *s*
`s + t`	Concatenación de *s* y *t*
`s[i]`	*i*-ésimo elemento de *s* (partiendo de $i = 0$)
`s[i:j]`	"rebanada" de *s*, desde *i* a *j*
`len(s)`	número de elementos de *s*
`min(s)`, `max(s)`	valor mínimo y máximo en *s*

Tipos de secuencias estándar

En Python existen tres tipos primitivos de secuencias:

Tupla Es una colección *fija* de elementos. Esto significa que no es posible agregar, modificar, o eliminar elementos una vez creada.

```
T = ( 1, 2, "tres", 4, "cinco" )
```

Lista Es una colección dinámica, es decir, sí es posible agregar y eliminar elementos, así como también modificarlos.

```
L = [ 1, 2, "tres", 4, "cinco" ]
```

Diccionario Es una asociación o mapeo de *claves* a *valores*, donde cada clave tiene un único valor asociado.

```
D = { 'lunes': 'fisica', 'martes': 'biologia',
      'miercoles': 'quimica',
      'jueves': 'matematica',
      'viernes': 'geologia' }
```

Tuplas

Como son *inmutables*, son frecuentemente usadas para "empaquetar" un grupo de valores y pasarlos como un sólo valor

```
phi = 0.5*pi
t = (cos(phi), sin(phi), sin(phi)*cos(phi))
# paso la tupla 't' como un solo argumento
ProcesarValores(t)

def ProcesarValores(t):
    # recibo la tupla 't' con
    # todos los elementos en ella
    print t[0]
    print t[1]
    print t[2]
```

Tuplas

De la misma manera, pueden ser usadas para hacer que una función retorne múltiples valores:

```
def CalcularFactores(v, c):
    return (v/c, sqrt(1.0-(v/c)**2))

(b, g) = CalcularFactores(1.0e+8, 3.0e+8)
print b
print g
```

La tupla retornada por `CalcularFactores` es *desempaquetada* al asignarla a una tupla de variables. La variable `b` recibe el primer elemento, v/c, mientras que `g` recibe el segundo, $\sqrt{1-(v/c)^2}$.

Listas

Al ser *modificables*, las listas tienen muchos más usos que las tuplas. Una lista puede ser usada como un arreglo con número fijo de elementos...

```
# guarda valores tabulados de y(x) en A
A = [ 0 for i in range(1000) ]
for i in range(1000):
    x = float(i)/1000.0 # x entre 0.0 y 1.0
    A[i] = y(x)
```

Pero también como un “arreglo dinámico” que se expande a medida que se agregan elementos:

```
# guarda valores tabulados de y(x) en A
A = [ ]
for i in range(1000):
    x = float(i)/1000.0
    A.append(y(x))
```

Operaciones con listas

Una lista soporta todas las operaciones definidas para una secuencia, más las siguientes:

`L[i] = x`	Reemplaza el elemento *i* de `L` por `x`
`L.append(x)`	Agrega el elemento `x` al final de `L`
`L.extend(s)`	Agrega cada elemento de `s` al final de `L`
`L.insert(i, x)`	Inserta `x` en la posición *i* en `L`
`L.count(x)`	Número de elementos iguales a `x`
`del L[i]`	Elimina el elemento en la posición *i* de `L`
`L.remove(x)`	Elimina los elementos iguales a `x` de `L`
`L.pop(i)`	Devuelve (y elimina) el elemento *i* de `L`
`L.reverse()`	Invierte la lista *en su sitio*
`L.sort(cmp,key,r)`	Ordena la lista *en su sitio*

Uso de listas

```
A = [ ]
B = [ 1, 2, 3 ]

A.insert(0, 42.0)
print A                 # -> [42.0]
A.append(17)            # agrega 17 al final
print A                 # -> [42.0, 17]

C = A + B               # concatena A y B
print C                 # -> [42.0, 17, 1, 2, 3]
del C[2]
print C                 # -> [42.0, 17, 2, 3]

C.sort()
print C                 # -> [2, 3, 17, 42.0]

print 17 in C           # -> True
print 53 in C           # -> False
```

Uso de listas

Otros usos de las listas:

- Como una “pila” (*stack*) de valores, donde sale el valor más reciente que entró:

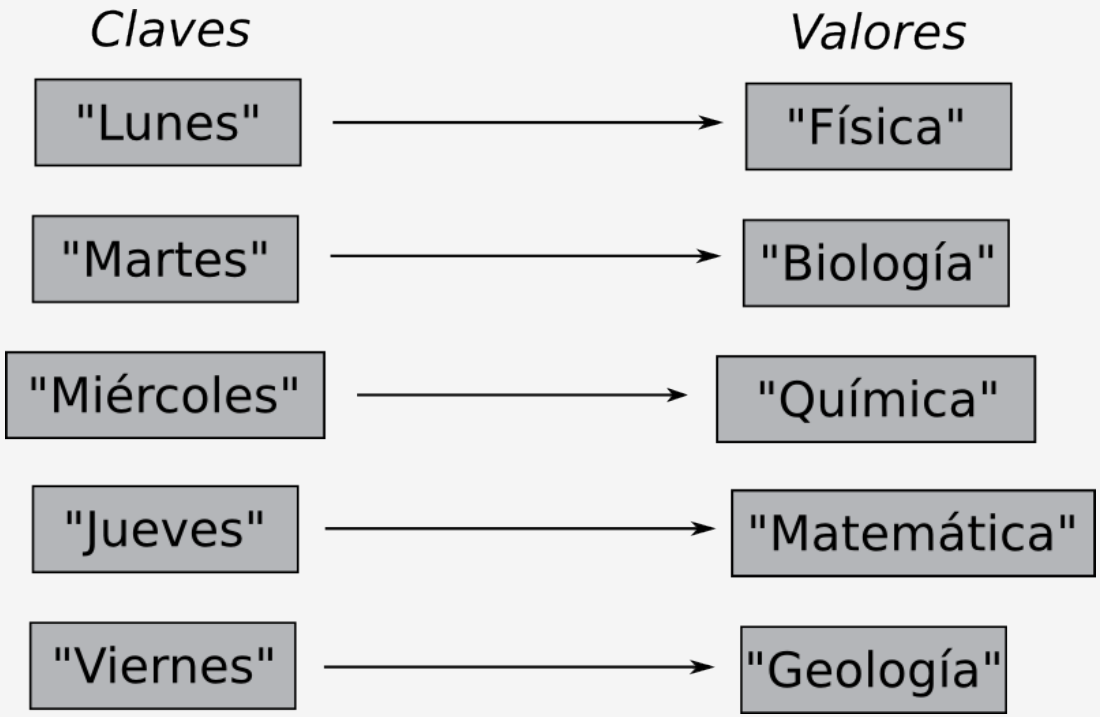
```
P = []
P.append(10.0) # [ 10.0 ]
P.append(20.0) # [ 10.0, 20.0 ]
P.pop()        # [ 10.0 ] -> 20.0
P.pop()        # [ ]      -> 10.0
```

- Como una “cola” (*queue*) de valores, donde los valores salen por “orden de llegada”:

```
Q = []
Q.append(10.0) # [ 10.0 ]
Q.append(20.0) # [ 10.0, 20.0 ]
Q.pop(0)       # [ 20.0 ] -> 10.0
Q.pop(0)       # [ ]      -> 20.0
```

Diccionarios

Un diccionario es un *mapeo* o *asociación* entre un conjunto de claves y un conjunto de valores, a cada clave unívocamente asignado un valor.



Un diccionario puede contener como valores cualquier objeto, pero sólo puede usar como claves ciertos tipos de objetos: `int`, `float`, `str`, `tuple`.

Diccionarios

```
# numeros mapea digitos
# a sus valores numericos
numeros = { 'uno': 1, 'dos': 2, 'tres': 3,
            'cuatro': 4, 'cinco': 5, 'seis': 6,
            'siete': 7, 'ocho': 8, 'nueve': 9 }
# vecinos mapea cada indice
# a una lista de indices vecinos
vecinos = { 0: [1, 2, 3], 1: [0, 2, 3, 7],
            2: [0, 1, 7, 3], 3: [0, 1, 2, 7] }

# pide el valor asociado a 'cinco'
print numeros['cinco']      # -> 5

# asigna nuevo valor a 'cinco'
numeros['cinco'] = 3.14159
print numeros['cinco']      # -> 3.14159
# elimina la clave 'cinco' y su valor
del numeros['cinco']
```

Operaciones con diccionarios

Un diccionario soporta todas las operaciones definidas para una secuencia, más las siguientes:

`D[k] = x`	Asigna el valor `x` a la clave `k`
`del D[k]`	Elimina la clave `k` (y su valor asociado)
`D.clear()`	Elimina todas las claves
`D.keys()`	Devuelve las claves de `D`
`D.values()`	Devuelve los valores de `D`
`D.get(k, v)`	(D[k] if k in D else v)
`D.setdefault(k, v)`	Lo mismo, pero hace `D[k] = v`

Algo sobre iteradores

- Tuplas, listas y diccionarios son secuencias estándar, integradas en el lenguaje mismo.
- Desde un punto de vista más general, una secuencia es cualquier objeto que puede ser *recorrido* elemento por elemento, por ejemplo con `for`.
- ¿Cómo es que es posible recorrer una tupla, una lista o un diccionario con `for`?
- Formalmente, las secuencias estándar pertenecen a una clase de objetos llamados **iteradores**, que cumplen con ciertos atributos y acciones (*métodos*).
- Es posible crear un *iterador* personalizado, imitando esos atributos y acciones (programación orientada a objetos). Este iterador puede ser recorrido en un `for` como si fuera una lista, tupla o diccionario, pero internamente puede generar valores dinámicamente.

Parte II

Lectura y procesamiento de archivos

Operaciones con strings

`s.strip(chars)`	Remueve `chars` de `s`
`s.rstrip(chars)`	Remueve `chars` del final de `s`
`s.lstrip(chars)`	Remueve `chars` del principio de `s`
`s.split(d)`	Lista de palabras en `s` delimitadas por `d`
`s.partition(d)`	(*antes de* `d`,`d`, *después de* `d`)
`s.startswith(chars)`	`True` si `s` comienza con `chars`
`s.endswith(chars)`	`True` si `s` termina con `chars`
`s.ljust(n)`	Justifica `s` a la izquierda en longitud `n`
`s.rjust(n)`	Justifica `s` a la derecha en longitud `n`
`s.center(n)`	Justifica `s` centrado en longitud `n`

Operaciones con strings

xterm

```
[sdavis@nostromo] ~> python
Python 2.5.2 (r252:60911, Nov 14 2008, 19:46:32)
[GCC 4.3.2] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> s = '   Nano Taller de Python    '
>>> print repr(s.rstrip())
'   Nano Taller de Python'
>>> print repr(s.lstrip())
'Nano Taller de Python    '
>>> print repr(s.strip())
'Nano Taller de Python'
>>> print s.split()
['Nano', 'Taller', 'de', 'Python']
>>> print s.partition('Taller')
('   Nano ', 'Taller', ' de Python    ')
>>> print repr('python'.center(10))
'  python  '
>>> print repr('python'.ljust(10))
'python    '
>>> print repr('python'.rjust(10))
'    python'
>>> print 'hola mundo'.startswith('hola')
True
>>> print 'hola mundo'.startswith('Hola')
False
>>> print 'hola mundo'.endswith('do')
True
>>>
```

Lectura de archivos

Para procesar las líneas una a una, sin almacenarlas, el patrón típico es:

```
for linea in file('archivo.txt'):
    # linea contiene la linea actual como string
    Procesar(linea)
```

Si se quiere leer todas las lineas de una vez, es posible usar el método `readlines()`, el cual devuelve una lista de strings:

```
lineas = file('archivo.txt').readlines()
```

Finalmente, para leer el archivo completo en un string, se usa el método `read()`.

```
texto = file('archivo.txt').read()
```

Ejemplo 1: lectura por columnas

La gran mayoría de las operaciones para desglosar (*parsear*) un archivo de texto se pueden conseguir con alguna combinación de `strip` y `split` sobre cada línea:

```
# lee la primera y la segunda columna
# del archivo como floats
for linea in file('tabla.dat'):
    # s: lista de columnas de la linea actual
    s = linea.strip().split()
    x, y = float(s[0]), float(s[1])
    print "%10f   %10f" % (x, y)
```

```
1     234 485 29                    1.000000  234.000000
2.    38  29485 28                  2.000000   38.000000
5     7  295                        5.000000    7.000000
6     4.32 580 102                  6.000000    4.320000
14    34  3.14 38 10 10            14.000000   34.000000
```

Ejemplo 2: lectura por clave = valor

```
param = { }  # diccionario vacio
for linea in file('parametros.txt'):
    linea = linea.strip()
    if linea.startswith('#') or linea == '':
        continue
    t = linea.partition('=')
    clave, valor = t[0].strip(), t[2].strip()
    param[clave] = valor
#
print param
```

```
# modo: sencillo/dificil
modo = sencillo
geometria = 2D
puntos = 137
# params. del modelo
alpha = 1.034
beta = 394.23
gamma = 20.0
```

```
{'puntos': '137',
'beta': '394.23',
'alpha': '1.034',
'modo': 'sencillo',
'gamma': '20.0',
'geometria': '2D'}
```

Lectura de archivos

A veces es necesario iterar un archivo de manera manual, para esto se leen las líneas una a una con `readline()` y se itera con `while`,

```
f = file('archivo.txt')
while True:
    linea = f.readline()
    if linea == '': break      # fin de archivo
    if ComienzaBloque(linea):
        while not TerminaBloque(linea):
            linea = f.readline()
            DentroDeBloque(linea)
    FueraDeBloque(linea)
```

Ejemplo 3: lectura por bloques

```
f = file('config.dat')
while True:
    linea = f.readline()
    if linea == '': break      # fin de archivo
    if linea.startswith('BEGIN'):
        print 'bloque', linea.split()[1]
        while not linea.startswith('END'):
            linea = f.readline()
            if linea.startswith('END'): continue
            print '   ', linea.split()
    else: print 'global:', linea.rstrip()
```

```
CELL 8.0 8.0 8.0
BEGIN atoms
   Ar 1.0 2.0 3.0
   Kr 5.0 3.4 7.9
END
ENERGY 9.47039
VOLUME 512.000
```

```
global: CELL 8.0 8.0 8.0
bloque atoms
    ['Ar', '1.0', '2.0', '3.0']
    ['Kr', '5.0', '3.4', '7.9']
global: ENERGY 9.47039
global: VOLUME 512.000
```

Ahora... a la práctica!

Problema 1:
Implementar un programa en Python que lea el siguiente formato de archivo:

```
[general]
 usuario = pedro
 usuario = juan
 usuario = diego

[pedro]
   quota = 5438
   used = 102
   blocked = no

...
```

Produciendo una salida como la siguiente:

```
Usuario 'diego' ha usado 74.31% de su cuota
Usuario 'juan' (bloqueado) ha usado 24.23% de su cuota
Usuario 'pedro' ha usado 1.88% de su cuota
```

Ahora... a la práctica!

Problema 2:
Implementar un programa en Python que lea el formato de archivo 'control' (de LPMD):

```
cell cubic 17.1191
input crystalfcc symbol=Ar nx=3 ny=3 nz=3
output xyz output.xyz each=10

use lennardjones
    sigma 3.4
    epsilon 0.001
enduse

use verlet
    dt 1.0
enduse

steps 1000
potential lennardjones Ar Ar
integrator verlet
```

Problema 2 (continuación)

Produciendo la siguiente salida:

```
La entrada es: crystalfcc
La salida es: xyz
El potential es: lennardjones
El integrador es: verlet
lennardjones: sigma=3.4, epsilon=0.001
verlet: dt=1.0
```

Hints:

- Ignore todo lo que no sea `input`, `output`, `potential`, `integrator`, `use`, `enduse`
- Use un diccionario para las opciones globales y un diccionario aparte para cada `use` que encuentre