Nano Taller de Python

Charla 2: “Programación estructurada”

Sergio Davis <sergdavis@gmail.com>

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

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

Esquema de trabajo

En la sesión anterior vimos:

- Las ideas detrás de Python
- Cómo editar y ejecutar programas en Python

Ahora veremos cómo:

- Mostrar información en la pantalla
- Formatear valores numéricos
- Usar variables
- Definir ciclos y condicionales
- Definir funciones y llamarlas
- Pasar y leer argumentos en funciones
- Retornar valores

Con esto deberíamos cubrir la mayor parte de la programación "tradicional", es decir BASIC, C, Fortran77, etc.

Parte I

Programación estructurada en Python

Salida a la pantalla

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.
>>> print "hola mundo"
hola mundo
>>> print 1,2,3
1 2 3
>>> print "uno", 2, "tres"
uno 2 tres
>>> print "El resultado es ", 2.71828
El resultado es  2.71828
>>> print int
<type 'int'>
>>> print abs
<built-in function abs>
>>> print (3 == 4)
False
>>>
```

La instrucción `print`

sintaxis de `print`

`print` *valor1* , *valor2* , ... , *valorN*
`print` *valor1* , *valor2* ,

- `print` agrega un salto de línea al final, a menos que termine con una coma
- `print` es una instrucción, no una función (no lleva paréntesis)
- `print` tiene otras sintaxis más complejas, que fueron eliminadas en Python 3, donde `print` sí es una función.

¿Qué pasa si queremos más control sobre la impresión? ¿Ancho de los campos? ¿Número de cifras significativas? ¿Notación científica?

Formatos de *string*

La mayor parte del tiempo la instrucción `print` es demasiado simple para usos científicos. En otros lenguajes existen "formatos" que permiten modificar la manera de mostrar un valor en la pantalla (`printf` en C, `FORMAT` en Fortran)

Como en Python un *string* es un objeto, quien se encarga de los formatos **es el *string* mismo**, usando el operador %

```
i = 5
z = 42.0
nombre = "posiciones"

print "El elemento %d de %s tiene el \
valor %f" % (i, nombre, z)
```

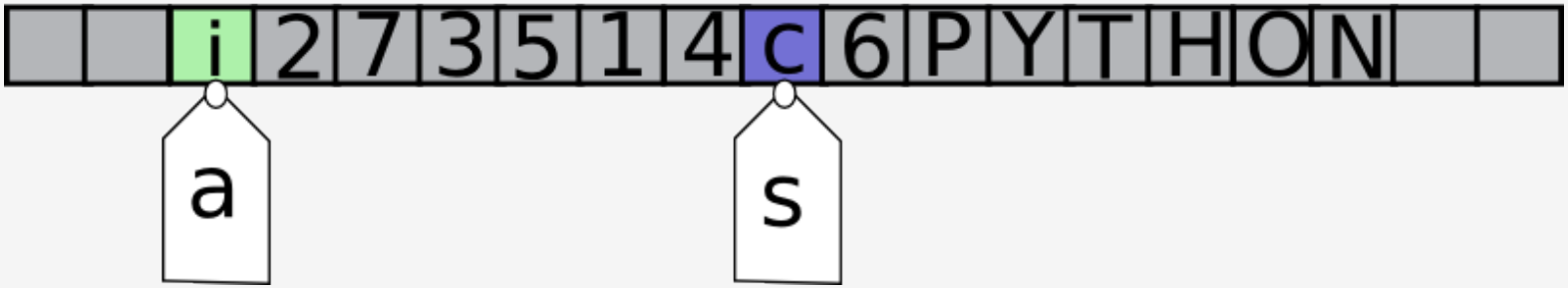
El elemento 5 de posiciones tiene el valor 42.000000

Formatos de *string*

Formato	Tipo	Ejemplo
`%d`	Entero	`"7"`
`%f`	Real	`"7.000000"`
`%.3f`	Real	`"7.000"`
`%08.3f`	Real	`"0007.000"`
`%e`	Real, notación científica	`"7.000000e+00"`
`%.3e`	Real, notación científica	`"7.000e+00"`

Estos formatos de *string* no sólo pueden ser usados en `print`, sino que en cualquier función que acepte un *string* como argumento.

Variables



En Python:

- Las variables son simples etiquetas para objetos en memoria
- Las variables no llevan un tipo asociado, los objetos sí
- Las variables pueden ser eliminadas explícitamente
- ***Cualquier objeto*** puede ser asignado a una variable. Esto incluye clases, funciones, módulos de Python, etc.

Es posible asignar múltiples variables a la vez:

```
a, b, c, d, e = 1, 2, 3, 4, 5
```

Más sobre variables

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.
>>> print x
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'x' is not defined
>>> x = 5
>>> print 7+x
12
>>> x = 42.0
>>> print 7+x
49.0
>>> x = 'hola mundo'
>>> print x
hola mundo
>>> print 7+x
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'int' and 'str'
>>> del x
>>> print x
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'x' is not defined
>>> x = 7
>>> x
7
>>>
```

Condicionales: `if`, `elif`, `else`

Ejecución de ciertos bloques de código según se cumplan o no determinadas condiciones

```
x = 5
if x < 3:
    print "El valor de x es menor que 3"
elif x < 6:
    print "El valor de x esta entre 3 y 6"
elif x < 9:
    print "El valor de x esta entre 6 y 9"
else:
    print "El valor de x es mayor o igual que 9"
```

Expresiones condicionales

Muy frecuentemente se necesita pasar o asignar un valor A cuando se cumple cierta condición y un valor B cuando no se cumple.

```
if p == 1:
    z = 10
else:
    z = 30
```

En Python este patrón se puede resumir con una expresión condicional:

```
z = (10 if p == 1 else 30)
```

La sintaxis de una expresión condicional es:

Sintaxis

(*valor*$_1$ `if` *condicion* `else` *valor*$_2$)

donde todo el paréntesis se evalúa a *valor*$_1$ si *condicion* es `True`, o *valor*$_2$ si *condicion* is `False`.

Usos de las expresiones condicionales

```
print "Hay %d %s en el banco" % \
  (N, ("peso" if N == 1 else "pesos"))
```

Para imprimir formas plurales

```
y = (x if x < 10.0 else (x**2)/10.0)
```

Para funciones por tramos

```
def Perim(radius): return 2*pi*radius
def Area(radius): return pi*radius**2
v = (Perim if q == 'perim' else Area)(1.0)
```

Para evaluar funciones condicionalmente

Ciclos

Idea de un ciclo o *loop*

Repetir una tarea un número prefijado de veces o hasta que se cumpla cierta condición.

Python tiene sólo **dos tipos** de ciclos:

`for` : Cuando se conoce de antemano cuántas veces se ha de repetir el ciclo

`while` : Cuando no se conoce de antemano cuántas veces se ha de repetir, sólo la condición de parada

Ciclo `for`

```
cubos = []
for i in range(7):
    ic = i**3
    cubos.append(ic)
    print ic, cubos
```

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

Más sobre `for`

La instrucción `for` no sólo itera sobre enteros: `for` itera sobre todos los elementos de una *secuencia*, asignando el valor del elemento a la variable. `range` es sólo una función conveniente que genera una lista de enteros.

```
for i in [ 3, 1, 4, 1, 5, 9, 2, 6, 5, 3 ]:
    print "Un digito de pi es", i

for i in [ 1, 2, 3, 4 ] + [ 3, 2, 1 ]:
    print i

for i in "estas son palabras al azar".split():
    print i
```

Ciclo `while`

```
from random import random
x = 0.5
while x < 0.6:
      x = random()
      print x
print "fin"
```

```
0.265077652566
0.598151187236
0.32932533897
0.136601311049
0.559332065265
0.501433130169
0.523576297001
0.126622354505
0.725716611695
fin
```

break y continue

La instrucción `break` interrumpe un ciclo `for` o `while`:

```
from random import random
i, s, limit = 0, 0.0, 15.0
while s < limit:  # termina si s >= limit
      s += random()
      i += 1
      if i > 30:
         break # tambien termina si i > 30
print s
```

break y continue

La instrucción `continue` hace que un ciclo `for` o `while` se salte a la iteración siguiente:

```
for i in range(15):
    if i % 2 == 0: continue
    print i, i+1
```

```
1 2
3 4
5 6
7 8
9 10
11 12
13 14
```

Funciones: `def`

Para modularizar un programa demasiado largo, un primer *approach* es el escribir ciertas porciones del programa como funciones. En Python, las funciones se declaran usando la instrucción `def`:

```
def MiFuncion():
    print "Esta es MiFuncion"

# aqui se llama a la funcion 5 veces
for i in range(5): MiFuncion()
```

Una función puede retornar un resultado usando la instrucción `return`:

```
def OtraFuncion():
    return 42

x = OtraFuncion()
print "El resultado de OtraFuncion() es", x
```

Paso de argumentos

Para que una función sea de verdad útil (y reutilizable), es necesario poder pasarle entradas. Los nombres de las entradas (o argumentos) que requiere una función se declaran a continuación del nombre en `def` (siempre entre paréntesis):

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

print FuncionSuma(5, 3)
print FuncionSuma(7, 42.0)
print FuncionSuma("hola", "mundo")
```

- Nunca se mencionan los tipos de datos de `x` e `y`, ni el tipo de datos que devuelve FuncionSuma
- Los argumentos y el valor retornado son, tal como las variables, simples etiquetas a zonas de memoria

¿Cómo chequear tipos de datos?

A veces puede ser de verdad necesario chequear tipos explícitamente (aunque no es para nada *pitónico*!).
Un truco más o menos claro para emular lenguajes estáticos es usar la función `type` y la instrucción `assert`:

```
def SumaEnteros(x, y):
    assert type(x) == int
    assert type(y) == int
    return x+y

print SumaEnteros(5, 3)     # -> 8
print SumaEnteros(7, 42.0) # -> AssertionError
```

La instrucción `assert` actúa como `pass` si la expresión que le sigue es verdadera, pero falla con `AssertionError` si es falsa.
Es más común usar `try` y `except` para "atrapar" el error si los tipos no son los adecuados.

Paso de argumentos con nombre

- Si la función que definimos tiene muchos argumentos, es fácil olvidar el orden en que fueron declarados.
- Como un argumento no lleva asociado un tipo, Python no tiene cómo saber que los argumentos están cambiados.
- Para evitar este tipo de errores, hay una manera de llamar a una función pasando los argumentos en cualquier orden arbitrario: se pasan usando el nombre usado en la declaración:

```
def Prueba(a, b, c):
    # %r formatea automaticamente cualquier valor
    print "a=%r, b=%r, c=%r" % (a, b, c)

Prueba(1, 2, 3)
Prueba(b=3, a=2, c=1)
```

```
a=1, b=2, c=3
a=2, b=3, c=1
```

Argumentos con valores por omisión

Para hacer que algunos argumentos sean opcionales, se les da valores por omisión en el momento de declararlos:

```
from math import sqrt

# argumento v es requerido, c es opcional
# c toma el valor 3.0e8 por omision
def Gamma(v, c = 3.0e+8):
    return sqrt(1.0-(v/c)**2)

print Gamma(0.1, 1.0)
print Gamma(1.e+7)     # usa c = 3.0e+8
```

Retorno de múltiples valores

Para hacer que una función retorne más de un valor, en lenguajes como Fortran, C o C++, lo que se hace es definir argumentos *de entrada* y argumentos *de salida*.
Para retornar múltiples valores en Python, lo usual es retornar los valores "empaquetados" en una *tupla*:

```
from math import atan, sqrt

def ModuloArgumento(x, y):
    norm = sqrt(x**2 + y**2)
    arg = atan2(y, x)
    return (norm, arg)

n, a = ModuloArgumento(3.0, 4.0)
print "Modulo es:", n
print "Argumento es:", a
```

Número variable de argumentos

¿Cómo hacemos que una función acepte un número no prefijado de argumentos? Es posible pasar una lista o tupla, pero Python ofrece una mejor solución:

```
def atan(*args):
    # args es una tupla de argumentos
    if len(args) == 1:
        return math.atan(args[0])
    else:
        return math.atan2(args[0], args[1])

print atan(0.2)           # 0.19739
print atan(2.0, 10.0)     # 0.19739
print atan(-2.0, -10.0) # -2.94419
```

Hay otra sintaxis para pasar argumentos pero como depende del uso de diccionarios la veremos en la siguiente sesión.

Manejo de errores

En Python, como en otros lenguajes, los errores de sistema se pueden manejar *interceptando excepciones*:

```
try: # intente hacer lo siguiente...
    f = file('archivo.txt')
    datos = f.read()
    f.close()
except IOError: # pero si ocurre IOError...
    print 'Error, no existe el archivo'
```

Esto es equivalente al `try/catch` de otros lenguajes.

Ahora... a la práctica!

La idea es aplicar todo lo visto hasta ahora en problemas reales, y así tener un dominio más o menos fluido en la forma *estructurada* de Python.

Problema 1:

Implementar un programa en Python que encuentre una solución positiva para $\cos(x) - x^3 = 0$ mediante el método de Newton-Raphson:

$$x_{n+1} = x_n + \frac{f(x_n)}{f'(x_n)}, \tag{1}$$

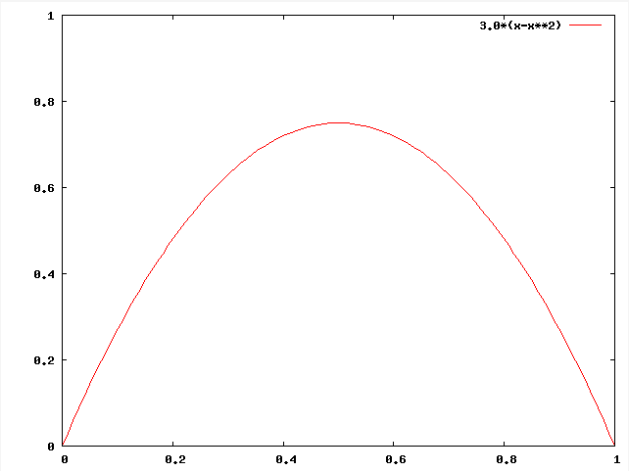
usando la suposición inicial $x = 0{,}5$.

La derivada se puede estimar numéricamente como

$$f'(x) = \frac{f(x+\Delta) - f(x-\Delta)}{2\Delta} \tag{2}$$

Ahora... a la práctica!

Problema 2:



Implementar un programa en Python que calcule la integral de $f(x) = 3(x - x^2)$ entre 0 y 1 por el método de Montecarlo, es decir... Generamos M puntos al azar en el plano que caigan en el cuadrado entre `(0, 0)` y `(1, 1)`, y contamos la cantidad de puntos N que caen bajo la curva $f(x)$. La integral se obtiene según

$$I = \int_0^1 f(x)dx = \frac{N}{M} \tag{3}$$

Hint: usar millones de puntos para una buena aproximación