Nano Taller de Python

Charla 4: “Programación Orientada a Objetos”

Sergio Davis <sergdavis@gmail.com>

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

13 de enero 2009, de 10:00 a 11:00

Parte I

Programación Orientada a Objetos

Esquema de trabajo

En la sesión anterior vimos cómo programar de manera estructurada en Python, y cómo trabajar con listas, tuplas y diccionarios. Si esto fuera todo lo que Python ofrece, no sería muy útil más que para pequeños scripts.
Ahora veremos cómo diseñar programas orientados al objeto en Python.

Con esto cubrimos el uso de Python como un lenguaje de programación para aplicaciones completas, equivalente a C++ o a Java.

¿Qué es orientación a objetos?

- En la programación que hemos visto hasta ahora, un programa se diseña pensando en **subrutinas** que el *computador* debe ejecutar, secuencialmente
- En programación orientada al objeto, un programa se diseña como un modelo donde un grupo de objetos, cada uno con **responsabilidades** que cumplir, interactúan para lograr una meta común
- El primer lenguaje orientado a objetos fue Simula 67, hoy en día los lenguajes más populares que soportan orientación al objeto son Java, C++ y Delphi
- Python por supuesto soporta esta técnica, en Python *todo es un objeto* con *atributos* y *métodos*.

Conceptos fundamentales

Clase Esquema abstracto de un tipo de objetos. Ej., un *Automóvil*

Objeto Un individuo o ejemplar particular de una clase. Ej., un *Fiat 600 rojo*

Atributo Una propiedad de un objeto. Ej., número de puertas, color, tipo de motor

Método Una acción que un objeto puede realizar. Ej., todo *Automóvil* puede acelerar, frenar, poner reversa, etc.

Herencia Una clase puede heredar métodos y atributos de una clase más general. Ej., la clase *Automóvil* hereda algunas de sus características de la clase *Vehículo*

Interfaz La apariencia externa de un objeto, definida por sus atributos y métodos públicos. El conjunto de las interfaces en una librería se denomina la API (Application Programming Interface).

Implementación La estructura interna del objeto, definida por sus atributos y métodos privados.

Ventajas de la orientación a objetos

- Es una forma muy clara de poner orden en la complejidad de un programa
- Naturalmente hace un programa modular, cada objeto es una unidad de código independiente
- Incentiva la **encapsulación**, es decir, una parte del código **no necesita saber** como funciona otra parte
- Permite la técnica del **polimorfismo**, es decir, programar algoritmos genéricos reutilizables en distintas situaciones.

¿Cómo definir una clase en Python?

```
def dist(a, b):
    """Distancia entre dos puntos en 2D"""
    return sqrt((a[0]-b[0])**2+(a[1]-b[1])**2)

class Circulo:
    """Circulo representa un circulo, con radio
       y centro. Sus atributos y metodos definen
       su API."""
    centro, r = (0.0, 0.0), 1

    def Diametro(self): return 2.0*self.r
    def Area(self): return pi*self.r**2
    def Perimetro(self): return 2.0*pi*self.r
    def EstaDentro(self, a):
        return (dist(a, self.centro) < self.r)
```

En Python, un método se diferencia de una función cualquiera en que lleva un argumento extra, por convención llamado `self`.

¿Cómo crear ejemplares de una clase en Python?

Simplemente llamamos al nombre de la clase, `Circulo`, como si fuera una función, y guardamos su resultado.

```
# c1 apunta a un objeto Circulo
c1 = Circulo()
# llenamos sus atributos
c1.r = 5.0
c1.centro = (0.3, 0.5)
# llamamos a sus metodos
print c1.Area()
print c1.Perimetro()
print c1.EstaDentro(3.0, 3.0)
# otro circulo
c2 = Circulo()
c2.r, c2.centro = 10.0, (3.0, 3.0)
print c2.Diametro()
print c2.EstaDentro(21.0, 17.0)
```

Constructores

Para no tener que crear el objeto y asignar sus atributos separadamente, se usan métodos especiales llamados *constructores*, que reciben argumentos. En Python el constructor siempre se llama `__init__`

```
class Circulo:
    centro, r = (0.0, 0.0), 1
    def __init__(self, r0, c0):
        """Constructor pasando radio y centro"""
        self.r, self.centro = r0, c0
    def Diametro(self): return 2.0*self.r
    def Area(self): return pi*self.r**2
    def Perimetro(self): return 2.0*pi*self.r
    def EstaDentro(self, a):
        return (dist(a, self.centro) < self.r)
# Construimos e inicializamos el objeto Circulo
# Python llama implicitamente a nuestro __init__
c1 = Circulo(5.0, (0.3, 0.5))
```

Atributos y métodos privados

C++ y Java permiten definir métodos y atributos **privados**, que sólo pueden ser vistos y modificados por el objeto mismo. Éstos esconden las “tuercas y engranajes” detrás del funcionamiento del objeto.

```
class DiaDelMes
{
 public:    // Interfaz
   Dia(): d(1) { }  // Constructor
   int Dia() { return d; }
   void AumentarDia()
   {
    d += 1;
    if (d > 31) d = 1;
   }
 private:   // Implementacion
   int d;   // atributo privado
}
```

Atributos y métodos privados en Python

En Python, a diferencia de C++ o Java, no existe el concepto de atributos o métodos privados. Esto por mantener el lenguaje simple, y porque...

> *"Una clase diseñada elegantemente no debería tener detalles feos que ocultar"*

Si de todas maneras se quieren atributos *privados*, basta con usar nombres que comiencen con dos *underscore*, como `__x`, que Python automáticamente renombra a `_Clase__x`.

```
class DiaDelMes:
    __d = 1

    def Dia(self): return self.__d
    def AumentarDia(self):
        self.__d += 1
        if self.__d > 31: self.__d = 1
```

Herencia

```
# Mamifero es nuestra clase base
class Mamifero:
    tieneCola, color = False, 'gris'
    def Gritar(self):
        pass
# Gato es un tipo especializado de Mamifero
class Gato(Mamifero):
    tieneCola, tieneCascabel = True, False
    nombre, color = 'Garfield', 'naranja'
    def Gritar(self):
        print self.nombre, 'dice Miau!'
# Perro tambien es un tipo de Mamifero
class Perro(Mamifero):
    tieneCola, nombre = True, 'Bobby'
    def Gritar(self):
        print self.nombre, 'dice Guau!'
```

Polimorfismo

```
# g es un ejemplar de la clase Gato
g = Gato()
g.nombre = 'jim'
g.color = 'pardo'
g.tieneCascabel = True
# p es un ejemplar de la clase Perro
p = Perro()
p.nombre = 'dexter'
p.color = 'cafe'
# Tanto g como p son ejemplares de Mamifero
# por lo tanto ''entienden'' el metodo Gritar
# pero cada uno lo implementa de manera distinta
for x in [ g, p ]: x.Gritar()
```

```
jim dice Miau!
dexter dice Guau!
```

¿Dónde está `virtual`?

```
class Mamifero:
    tieneCola, color = False, 'gris'
    def Gritar(self):
        pass
```

Los programadores de C++ echarán de menos la declaración `virtual` en el ejemplo anterior...

```
class Mamifero
{
 public:
  bool TieneCola;
  std::string color;

  Mamifero(): tieneCola(false), color("gris") { }
  virtual void Gritar() { };
};
```

Polimorfismo en Python

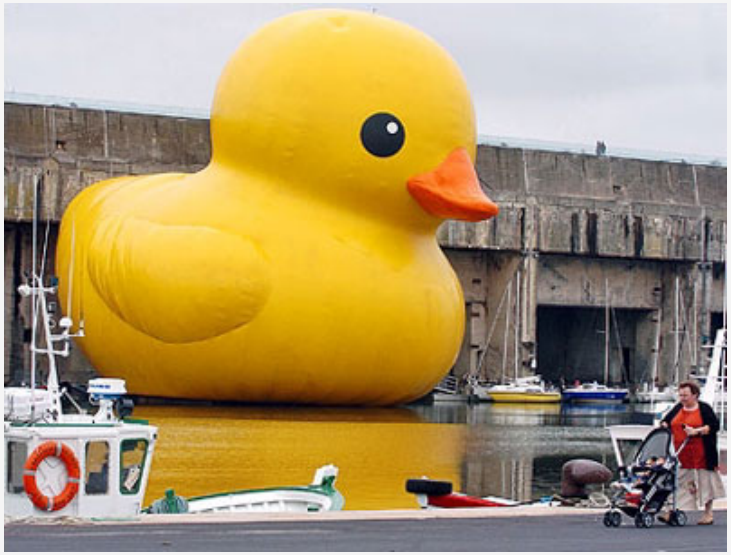
En C++ deben declararse en la clase base con la palabra clave `virtual` todos los métodos que se espera sean redefinidos en las clases más especializadas.
Python en este sentido es más como Java, todos los métodos pueden ser redefinidos (o sea son *por omisión* `virtual`).
Existen también métodos completamente abstractos, que no tiene sentido definirlos (ni siquiera vacíos) en una clase base...

> *¿Cómo sería el método `CalcularPerimetro()` de la clase `FiguraGeometrica`?*

Uno está obligado a definir estos métodos en las clases especializadas. A éstos en C++ y Java se les llama métodos **virtuales puros**.

Python no tiene métodos virtuales puros. La manera *pitónica* de conseguir polimorfismo es un ingenioso concepto llamado *duck typing*.

Duck Typing



“Si camina como un pato y hace quack como un pato, yo lo llamaría un pato. . . ”

No importa si una clase deriva o no de otra, lo realmente importante es cómo se ve por fuera (su *interfaz*). Una clase debería poder *hacerse pasar* por otra si imita los los mismos atributos y métodos.

No es necesaria la herencia para conseguir polimorfismo, lo que permite una flexibilidad equivalente a los *templates* de C++, pero sin la complicación de una nueva sintaxis.

Duck typing versus *templates*

```
template <class T> T & MayorArea(T & a, T & b)
{
 if (a.Area() > b.Area()) return a;
 else return b;
}
// Tanto Cuadrado como Circulo derivan de Figura
Cuadrado a(5.0);
Circulo b(22.0);
std::cout << MayorArea<Figura>(a, b).Perimetro();
std::cout << std::endl;
```

```
def MayorArea(a, b):
    return (a if a.Area() > b.Area() else b)
# Tanto Cuadrado como Circulo definen Area()
a = Cuadrado(5.0)
b = Circulo(22.0)
print MayorArea(a, b).Perimetro()
```

Set y Get

En C++ existe un patrón de programación muy frecuente, el abuso de los métodos `Set` y `Get`:

```
class Foo
{
 public:
   double GetX() { return x; } // metodo Get
   void SetX(double x0)        // metodo Set
   {
    if (x0 < 0) x = 0.0;
    else x = x0;
   }
 private:
   double x;
};
Foo f;
f.SetX(42.0); // f.x = 42.0
// f.x = -f.x+sqrt(f.x)
f.SetX(-f.GetX()+sqrt(f.GetX()));
```

Propiedades

Python permite ocultar los `Get` y `Set` dentro de un atributo `property` que se comporta como si fuera una variable de verdad:

```
class Foo:
    __x = 0.0
    def __getx(self): return self.__x
    def __setx(self, x):
        self.__x = (0.0 if x < 0.0 else x)
    # property funde los getx y setx
    # en un atributo transparentemente
    x = property(__getx, __setx)

f = Foo()
f.x = 42.0
f.x = -f.x + sqrt(f.x)
```

Manejo de errores: Excepciones

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.

Cómo arrojar excepciones personalizadas

```
# Nuestra excepcion es una clase vacia
class ErrorValorNegativo: pass

def raizcuadrada(x):
    if x < 0.0:
        # arroja la excepcion
        raise ErrorValorNegativo()
    else:
        return sqrt(x)

try:
    print raizcuadrada(-1.0)
except ErrorValorNegativo:
    # atrapa nuestra excepcion
    print 'El numero no puede ser negativo!'
```

Sobrecarga de operadores

En Python es posible sobrecargar los operadores, tal como en C++, lo cual, entre otras cosas, es útil para asignar nuevos significados a las operaciones matemáticas.
Simplemente se declaran nuevos métodos con los mismos nombres que los operadores por omisión:

```
class Complex:
    re, im = 0.0, 0.0

    def __init__(self, re, im):
        self.re, self.im = re, im
    def __str__(self):
        return '<%f + i*%f>' % (self.re, self.im)
    def __add__(self, x):
        self.re += x.re
        self.im += x.im
    def __sub__(self, x):
        self.re -= x.re
        self.im -= x.im

a = Complex(1.0, 0.0)
b = Complex(2.0, 3.0)
print a+b
print a-b
```

Algunos operadores en Python

Todos estos operadores empiezan y terminan con doble *underscore*

init	Constructor
del	Destructor
str	Impresión con `print`
add	Suma
sub	Resta
mul	Multiplicación
div	División
lt	Menor que
gt	Mayor que
eq	Igualdad, ==
ne	Desigualdad, !=
call	Llamada como función

Ahora... a la práctica!

Problema:
Implementar en Python una clase `Matrix`, con la siguiente interfaz:

- Constructor pasando número de filas y columnas
- `Rows()` devuelve el número de filas
- `Columns()` devuelve el número de columnas
- `Get(columna, fila)` devuelve el valor de una celda
- `Set(columna, fila, valor)` asigna el valor a una celda
- `SetLabel(columna, label)` asigna el nombre a una columna
- `GetLabel(columna)` devuelve el nombre de una columna
- Operadores + y - suman y restan matrices
- Al imprimir una matriz, muestre los nombres de las columnas como encabezado

(Esta es la interfaz de la clase `lpmd::Matrix` en LPMD)