Python Avanzado

Ernesto Revilla

Software Integrado para el Control de Empresas, s.l.

GCubo: Grupo de usuarios de GNU/Linux de Granada

E.T.S.Ing. Informática (Univ. Granada)

Abril 2.004

Herencia simple desde tipos internos

Python, desde la versión 2.2 permite heredar desde tipos de datos internos, como:

- int, float, complex, str, unicode
- list, dict, tuple

```
class DefaultDict(dict):
    def __init__(self, initialValues=None, default=0.0):
        dict.__init__(initialValues or {})
        self._default=default
    def __getitem__(self, key):
        try:
            return dict.__getitem__(self, key)
        except KeyError:
            return self._default
if __name__=="__main__":
    d=DefaultDict({'a': 5})
    print 'valor de e:', d['e']
    print 'valor de a:', d['a']
```

Vea DefaultDicc.py

Simular tipos internos 1

Existen construcciones sintácticas que facilitan la redacción de los programas, como [ ], { }, !=, ... Podemos personalizar su comportamiento.
Realizamos un diccionario por delegación:

```
class MiDicc:
    def __init__(self, diccInicial=None):
        self._dicc=dict(diccInicial or {})
    def __getitem__(self, item):
        print "leyendo", item
        return self._dicc[item]
    def __setitem__(self, item, valor):
        print "estableciendo", item
        self._dicc[item]=valor
    def zap(self):
        self._dicc={}
    def __getattr__(self, attr):
        if attr in ['items', 'keys', 'values']:
            return getattr(self._dicc, attr)
```

Vea MiDicc.py.

Simular tipos internos 2

Usar las entradas del diccionario como atributos:

```
class MiDicc2(MiDicc):
    def __getattr__(self, attr):
        return MiDicc.__getitem__(self, attr)
    def __setattr__(self, attr, valor):
        MiDicc.__setitem__(self, attr, valor)
```

produce bucle infinito!!, al establecer atributo _dicc in MiDicc

```
    def __setattr__(self, attr, valor):
        if attr not in ['_dicc']:
            MiDicc.__setitem__(self, attr, valor)
        else:
            MiDicc.__dict__['_dicc']=valor
            # si MiDicc hubiese heredado de object
            # MiDicc.__setattr__(self, attr, valor)
```

Vea MiDicc2.py.

Un ejemplo práctico

Para este ejemplo es necesario tener la base de datos sqlite (www.sqlite.org), como también el binding para python (pysqlite.sourceforge.net).
Conexión a una bd sqlite:

```
import sqlite
con=sqlite.connect('archivo.sqlite')
cur=con.cursor()
def crearTabla():
  cur.execute("create table personas (num integer, "\
      "nombre text, "apellidos text, edad integer;")
def insertarDatosEjemplo():
  personas=[ [1, 'Erny', 'Revilla'], [2, 'Lorenzo', 'Gil'],
             [3, 'Dani', 'Molina']]
  for persona in personas:
    cur.execute("insert into personas (num, nombre, "\
     "apellidos) values (%s, '%s', '%s');" % tuple(persona))
```

Vea los archivo db1.py, db2.py y db3.py que muestran como puede crearse una pequeña capa de persistencia para objetos.

Cosas sobre __init__

- __init__ representa el método de inicialización de instancia, no es el constructor (como en c++) sino el inicializador.
- __init__ es llamado después de haberse reservado la memoria y realizado una minima inicialización del objeto.
- al realizar herencia múltiple es necesario definir __init__ para indicar el orden de las llamadas de __init__ de las superclases

```
class A(object):
    def __init__(self):
        print "__init__ de A"
class B(object):
    def __init__(self):
        print "__init__ de B"
class C(A,B): pass   # esto está mal!!!
c=C()
class C(A,B):
    def __init__(self):
        A.__init__(self)
        B.__init__(self)
```

Vea EjemploInit.py

__new__, el constructor

- __new__ siempre es un método de clase
- __new__ aparece en la versión 2.2 de python
- con __new__ se puede devolver también instancias de otros tipos.

Ejemplos:

```
class A(object):
    def __init__(self, *args, **kwargs):
        print "creando objeto de tipo A con "\
              "argumentos pos. %s y argumentos nombrados %s"
   % (args, kwargs)

# devolver otros tipos de objetos
class B(object):
    def __new__(cls, *args, **kwargs):
        return A(*args, **kwargs)
x=B(1,2, pepe='no')
```

Vea EjemploNew.py

__new__: singleton

```
# singleton:
class Singleton(object):
    _instance=None
    def __new__(cls, *args, **kwargs):
        if not cls._instance:
            print "Creando Singleton"
            cls._instance=object.__new__(cls, *args,
  **kwargs)
        return cls._instance

    def __init__(self, *args, **kwargs):
        print "Inicializando Singleton"

s=Singleton()
s=Singleton()
```

Metaclases

- Al terminar la lectura de la definición de una clase, Python crea ésta usando una metaclase de forma Metaclase(nombre, superclases, diccionario)
- Nombre es el nombre de la clase, Superclases es una tupla
- diccionario contiene las funciones e identificadores definidos dentro de la clase
- Actualmente (>=2.2) usa dos metaclases, una para las clases antiguas (types.ClassType) y otra para las nuevas (type)
- En la metaclase podemos definir nuevos atributos o funciones para la clase

```
class Metaclase(type):
      def __init__(self, name, bases, dic):
         type.__init__(self, name, bases, dic)
         print "\nDefiniendo clase %s:" % name
         print "Bases: %s" % (bases or None)
         print "Dict: %s" % dic
class A(object):
      __metaclass__=Metaclase
      _attributes=["at1","at2","at3"]
      y=5
      def __init__(self):
         self.x=10
```

Metaclases 2

Vea Metaclases.py para ejemplo anterior.

El siguiente ejemplo crea nuevos métodos en una clase (getters y setters) para los atributos especificados.

```
class Metaclase(type):
   def __init__(self, name, bases, dic):
      type.__init__(self, name, bases, dic)
      if "_attributes" in dic:
         for attr in dic['_attributes']:
            f=lambda s,a=attr: getattr(s,a)
            setattr(self, 'get'+attr, f)
            f=lambda s, value, a=attr: setattr(s,a,value)
            setattr(self, 'set'+attr, f)
```

Vea Metaclases2.py