Programación orientada a objetos

¿Qué es eso?

Nos introducimos ahora en lo que cinco años atrás se consideraba un tema muy avanzado. Actualmente la *Programación orientada a objetos* se ha convertido en la norma. En algunos lenguajes como Java o Python este concepto tiene tanta aplicación que resulta difícil no toparse con algún objeto por más simple que sea nuestro programa. Pero, en definitiva, ¿de qué se trata?

En mi opinión, las mejores introducciones son:

- *Object Oriented Analysis* por Peter Coad & Ed Yourdon.
- *Object Oriented Analysis and Design with Applications* por Grady Booch (si podés encontrar la primera edición)
- *Object Oriented Software Construction* por Bertrand Meyer (tratá de conseguir la segunda edición)

El orden de estos libros es de acuerdo a profundidad, complejidad y exactitud crecientes. Para la mayor parte de los programadores no profesionales el primero es el más adecuado. Para una introdución más focalizada en la programación recomiendo *Object Oriented Programming* de Timothy Budd (2a edición). No he leído este último, pero ha sido recomendado por profesionales cuyas opiniones respeto. Por último, podés probar el sitio http://www.cetus-links.org donde hay mucha información sobre la orientación a objetos.

Dando por sentado que no tenés tiempo ni ganas para leer todos estos libros o visitar los links, haré ahora una breve presentación del tema. (**Nota:** Algunos consideran a la programación orientada a objetos (POO) como algo muy complicado, otros la comprenden de entrada. No te preocupes si te encontrás entre el primer grupo, igualmente podrás utilizar los objetos sin haber comprendido del todo la idea que los subyace).

Una última aclaración: en esta sección usaremos únicamente Python ya que tanto BASIC como Tcl no soportan objetos. Es posible implementar una diseño orientado a objetos en un lenguaje no orientado a objetos a partir de ciertas convenciones de codificación, pero no siempre es una buena idea y es poco recomendable. Si para resolver un problema encontrás que la mejor solución se halla a partir de un diseño orientado a objetos, lo mejor será utilizar un lenguaje que permita trabajar facilmente con dichas técnicas.

Datos y funciones todos juntos

Los objetos son colecciones de datos y funciones que operan sobre esos mismos datos. Ambos se agrupan en un todo de tal manera que uno puede pasar un objeto desde una parte del programa a otra, teniendo de esta manera acceso no sólo a los datos (*atributos*) sino también a las funciones (*operaciones*) que están disponibles en el objeto.

Por ejemplo, un objeto de cadena almacena una cadena de caracteres, pero también provee una serie de *métodos* que operan sobre dicha cadena, tales como calcular el largo, pasar a mayúsculas, etc.

Los objetos usan la metáfora de la *mensajería* por medio de la cual un objeto pasa un mensaje a otro objeto y este objeto receptor responde ejecutando una de sus operaciones, es decir, un método. De esta manera un objeto es *invocado* al recibirse el mensaje correspondiente por parte del objeto que lo posee. Se utilizan diversas notaciones para representar esto, pero la más común es la que imita el acceso a los campos de un registro, es decir, la utilización de un punto. Así, si utilizamos una clase de elementos gráficos ("widget"):

```
w = Widget() # creamos w, una nueva instancia de Widget
w.pintar()  # le enviamos el mensaje "pintar"
```

Esto produce que el método "pintar" del objeto widget sea invocado.

Definiendo una clase

De la misma forma que los datos tienen diferentes tipos, los objetos también pueden tener distintos tipos. Estas colecciones de objetos de características idénticas se denominan colectivamente como una *clase*. Podemos definir clases y crear *instancias* de ellas, que son los objetos actuales con los cuales nos manejaremos. Podemos a su vez almacenar en variables de nuestros programas referencias a estos objetos para ser reutilizados.

Veamos un ejemplo concreto para mejorar un poco esta explicación. Crearemos una clase "de mensajería" que contendrá una cadena (el texto del mensaje) y un método que será el encargado de mostrar el mensaje.

```
class Mensaje:
    def __init__(self, cadena):
        self.texto = cadena
    def mostrar(self):
        print self.texto
```

Nota 1: Uno de los métodos de esta clase se define como `__init__` y es un método especial que recibe la denominación de *constructor*. La razón de este nombre radica en que dicho método es invocado automáticamente cuando se crea una nueva instancia del objeto. Cualquier variable creada dentro de este método será única y propia de la nueva instancia creada. En Python hay una serie de métodos de este tipo que llevan como identificador la estructura `__xxx__`

Nota 2: Ambos métodos definidos tienen un primer parámetro `self`. Este nombre es una convención que indica la instancia del objeto. Como veremos más adelante este parámetro es completado en tiempo de ejecución por el intérprete por lo cual no debemos preocuparnos nosotros. Por eso, en realidad nuestro método "mostrar" es invocado sin argumentos `m.mostrar()`.

Nota 3: A esta clase le pusimos el nombre `Mensaje` con M mayúscula. Esta es una convención, pero que en general es bastante respetada, no sólo en Python sino también en otros lenguajes orientados a objetos. Esta misma convención establece que los métodos

deben nombrarse con la inicial en minúscula y las subsiguientes palabras con sus iniciales en mayúscula. De esta forma un método que calcule el balance actual podría llamarse `calcularBalanceActual`.

Quizás quieras repasar la sección dedicada a los 'Datos' y revisar lo que dijimos acerca de los tipos definidos por el usuario. El ejemplo incluido en dicha sección debería resultarte ahora más claro. Esencialmente el único tipo definido por el usuario en Python es la clase. Una clase con atributos pero sin métodos es equivalente a un registro de BASIC.

Usando clases

Una vez que hemos definido la clase podemos crear instancias de ella y manipularlas:

```
m1 = Mensaje("Hola mundo")
m2 = Mensaje("Adiós, fue breve pero bueno")

note = [m1, m2] # ponemos los objetos en una lista
for msg in note:
    msg.mostrar() # mostramos cada mensaje por turnos
```

Esencialmente lo que hacemos es tratar a la clase como si fuera un tipo de datos estándar de Python.

Lo mismo pero distinto

Hasta aquí hemos visto la posibilidad de definir nuestros propios tipos (clases), crear instancias de ellos y asignarlos a variables. Luego también podemos enviarles a estos objetos mensajes que provocarán la ejecución de los métodos que definimos previamente. Sin embargo, aun resta un elemento muy importante de la Programación Orientada a Objetos, que en muchos aspectos es el más importante de todos.

Supongamos que tenemos dos objetos de clases diferentes pero que presentan el mismo tipo de mensajes a partir de sus propios y correspondientes métodos. En este caso podríamos juntar ambos objetos y tratarlos de manera idéntica en nuestro programa, pero sin embargo, los objetos se comportarán según su forma inicial. ¿Extraño no? Bueno, esta habilidad de reaccionar de manera diferente frente a los mismos mensajes de entrada se denomina *polimorfismo*.

Podríamos utilizar esto para obtener una serie de objetos gráficos diferentes que se dibujaran automáticamente al recibir el mensaje "pintar". Obviamente no es lo mismo dibujar un círculo que un triángulo, pero si ambos tienen un método "pintar" podemos ignorar las diferencias y tratarlos como "figuras".

Veamos un ejemplo en el cual en lugar de dibujar las figuras calculamos su superficie:

Primero creamos las clases Cuadrado y Círculo:

```
class Cuadrado:
    def __init__(self, lado):
        self.lado = lado
    def calcularSuperficie(self):
        return self.lado**2

class Circulo:
    def __init__(self, radio):
        self.radio = radio
    def calcularSuperficie(self):
        import math
        return math.pi*(self.radio**2)
```

Nota: Obtuvimos el valor de pi del módulo math. Es frecuente encontrar constantes ya definidas en módulos como este.

Ahora podemos crear una lista de figuras (círculos o cuadrados) y luego imprimir sus superficies:

```
lista = [Circulo(5),Circulo(7),Cuadrado(9),Circulo(3),Cuadrado(12)]

for figura in lista:
    print "La superficie es: ", figura.calcularSuperficie()
```

Si combinamos estas ideas con los módulos podemos obtener un mecanismo muy poderoso para reutilizar nuestro código. Si colocamos las definiciones de nuestras clases en un módulo ("figuras.py" por ejemplo), cuando necesitemos manipular superficies bastará con importar dicho módulo. Esto se ha hecho con muchos de los módulos estándar de Python. Por esta razón, acceder a los métodos de un objeto se parece mucho a utilizar funciones en un módulo.

Herencia

La herencia se utiliza con frecuencia como un mecanismo para implementar el polimorfismo. En muchos lenguajes orientados a objetos la herencia es la única forma de implementar el polimorfismo. Este mecanismo funciona así:

Una clase puede *heredar* tanto los atributos como las operaciones de una clase *madre*, también llamada *super clase*. Esto implica que una nueva clase que sea bastante similar a otra clase en varios aspectos no necesita reimplementar nuevamente todos los métodos de esa segunda clase, sino que puede heredar de ella los métodos y sobreescribir aquellos que son diferentes (como en el caso del ejemplo anterior con las figuras del círculo y el cuadrado).

Nuevamente un ejemplo ilustrará mejor tanta teoría. Usaremos una *jerarquía de clases* de cuentas bancarias donde podremos depositar en efectivo, obtener el saldo y retirar dinero. Algunas cuentas nos darán interés (el cual, para nuestros propósitos, se calculará sobre cada depósito - una innovación interesante para el mundo financiero!) y otras nos cobrarán una tarifa por realizar extracciones de fondos.

La clase CuentaBancaria

Veamos cómo funcionará esto. Primero vamos a considerar los atributos y las operaciones típicas de una cuenta bancaria en un nivel abstracto.

En general es mejor considerar primero las operaciones que se pueden hacer y luego agregar los atributos necesarios para poder llevar a cabo estas operaciones. Entonces, en una cuenta bancaria podemos

- *Depositar* efectivo,
- *Retirar* efectivo,
- *Revisar nuestro saldo* y
- *Transferir* fondos a otra cuenta.

Para llevar adelante estas operaciones necesitaremos un número de cuenta bancaria (para las transferencias) y el saldo actual.

Podemos crear una clase que realice esto.

```
SaldoError = "Lo siento, sólo tiene $%6.2f en su cuenta"

class CuentaBancaria:
    def __init__(self, montoInicial):
       self.saldo = montoInicial
       print "Cuenta creada con un saldo de %5.2f" % self.saldo

    def deposito(self, monto):
       self.saldo = self.saldo + monto

    def extraccion(self, monto):
       if self.saldo >= monto:
          self.saldo = self.salto - monto
       else:
          raise SaldoError % self.saldo

    def checkSaldo(self):
       return self.saldo

    def transferencia(self, monto, cuenta):
       try:
          self.extraccion(monto)
          cuenta.deposito(monto)
       except SaldoError:
          print SaldoError
```

Nota 1: Revisamos el saldo antes de extraer el dinero y el uso de excepciones para manejar los errores. Obviamente no existe un tipo de error `SaldoError` así que lo creamos nosotros: es simplemente una variable de cadena!

Nota 2: El método `transferencia` utiliza los métodos `extraccion / deposito` de la clase `CuentaBancaria` para realizar su cometido. Esta técnica es muy común en la programación orientada a objetos y se denomina *autollamada*. Esto implica que las *clases derivadas* pueden implementar sus propias versiones de los métodos `deposito/extraccion` pero el método `transferencia` permanecerá igual para todos los tipos de cuentas bancarias.

Nota 3: Utilizamos el especificador de formato `%6.2f` para la cadena SaldoError. Este define un formato de seis caracteres con dos dígitos después de la coma.

La clase CuentaInteres

Ahora utilizaremos la herencia para crear una cuenta que sume el 3% de interés en cada depósito. Será idéntica a nuestra clase estándar CuentaBancaria a excepción del método "deposito". Simplemente lo "sobreescribimos":

```
class CuentaInteres(CuentaBancaria):
   def deposito(self, monto):
       CuentaBancaria.deposito(self, monto)
       self.saldo = self.saldo * 1.03
```

Y eso es todo. Ahora empezamos a comprender el poder de la programación orientada a objetos: todos los demás métodos fueron heredados de la clase CuentaBancaria (al haber puesto `CuentaBancaria` entre paréntesis luego del nombre de la nueva clase). Notá también que el nuevo método deposito llama al método deposito de la superclase en vez de copiar nuevamente todo el código. Esto tiene a su vez otra ventaja: si el día de mañana modificamos el método deposito de la clase CuentaBancaria (agregándole otras funcionalidades por ejemplo), la *subclase* recibirá estos cambios automáticamente sin necesidad de reescribir el código.

La clase CuentaRecargo

Esta cuenta es idéntica a nuestra clase CuentaBancaria, salvo que agrega un recargo de $3 por cada extracción de fondos. De la misma manera que con la clase CuentaInteres creamos una clase que heredará los métodos de CuentaBancaria y escribimos un nuevo método extracción.

```
class CuentaRecargo(CuentaBancaria):
    def __init__(self, montoInicial):
        CuentaBancaria.__init__(self, montoInicial)
        self.recargo = 3

    def extraccion(self, monto):
        CuentaBancaria.extracción(self, monto+self.recargo)
```

Nota 1: Guardamos el recargo como una variable de instancia para poder modificarla si es necesario más adelante. Notá también que podemos llamar al método `__init__` heredado de la misma manera que a cualquier otro método.

Nota 2: Simplemente agregamos el recargo al monto de extracción deseado y llamamos al método extraccion de la clase CuentaBancaria para que efectúe la operación.

Nota 3: Aquí introducimos un efecto de rebote: el recargo se aplicará automáticamente tanto en las extracciones como en las transferencias, pero quizás esa era nuestra intención, así que no hay mayores problemas.

Probando nuestro sistema

Para comprobar que nuestro sistema funciona, probá ejecutar el siguiente código en la línea de comando de Python o creando un archivo separado.

```
from cuentabancaria import *

# Primero una cuenta bancaria estándar
a = CuentaBancaria(500)
b = CuentaBancaria(200)
a.extraccion(100)
# a.extraccion(1000)
a.transferencia(100,b)
print "A = ", a.checkSaldo()
print "B = ", b.checkSaldo()


# Ahora una cuenta con interés
c = CuentaInteres(1000)
c.deposito(100)
print "C = ", c.checkSaldo()


# Luego una cuenta con recargo
d = CuentaRecargo(300)
d.deposito(200)
print "D = ", d.checkSaldo()
d.extraccion(50)
print "D = ", d.checkSaldo()
d.transferencia(100,a)
print "A = ", a.checkSaldo()
print "D = ", d.checkSaldo()


# Finalmente transferimos desde una cuenta con recargo a una con interés
# La del recargo deberá cobrarnos el recargo y la cuenta de interés
# deberá acreditarnos el interés
print "C = ", c.checkSaldo()
print "D = ", d.checkSaldo()
d.transferencia(20,c)
print "C = ", c.checkSaldo()
print "D = ", d.checkSaldo()
```

Ahora sacále el comentario a la línea `a.extraccion(1000)` para ver si la excepción funciona bien.

Bueno, esto ha sido todo. Un ejemplo razonable y bien directo del funcionamiento de la herencia y cómo esta puede ser utilizada para ampliar un esquema básico con nuevas y poderosas funciones.

Hemos visto cómo podemos construir nuestro programa en diferente pasos y cómo crear un programa para comprobar su funcionamiento. Nuestras rutinas de comprobación no están completas ya que no hemos cubierto cada uno de los posibles casos y todavía quedan varias comprobaciones para hacer, tales como qué sucede si creamos una cuenta con un monto negativo, etc.

Espero que este ejemplo te haya dado una idea de la programación orientada a objetos y te despierte el interés por conocer más acerca de ella por medio de los tutoriales o los libros que te he recomendado. Usaremos nuevamente la programación orientada a objetos en las restantes secciones y específicamente en el "Caso de Estudio".

Anterior Próxima Contenido

Si tenés sugerencias o dudas podés enviar un email en inglés a: alan.gauld@btinternet.com o en español a: manilio@zoommail.com