Programación Orientada a Objetos con Python

Tabla de contenidos

Conceptos Generales....2
El paradigma orientado a objetos....2
Clase....2
Objeto (Instancia)....2
Propiedades....3
de Clase....3
de Instancia....3
Métodos....3
El parámetro "self"....3
Sobrecarga de operadores....4
Constructor y Destructor....5
Getter / Setter / Deleter....5
Características Fundamentales....7
Encapsulamiento....7
Herencia....7
Herencia Múltiple....8
Orden de Resolución de Métodos....8
Polimorfismo....10

Conceptos Generales

La programación orientada a objetos (POO) es un paradigma de programación que representa conceptos en una estructura denominada clase, que integra a las propiedades (datos) que describen al concepto, y a los procedimientos asociados, conocidos como métodos.

Los objetos se crean a partir de las clases, y pueden interactuar entre sí mediante mensajes. Se puede considerar a un objeto como compuesto de sustantivos (variables) y verbos (funciones).

Smalltalk, C++, Objective-C, C#, Java, Javascript y Python son sólo algunos ejemplos de lenguajes de programación orientados a objetos.

El paradigma orientado a objetos

Un programa orientado a objetos puede verse como una colección de objetos que interactúan entre sí, en oposición a la visión del modelo convencional, en la que se ve un programa como una lista de tareas (subrutinas) para llevar a cabo.

```
class MiClase:                          # Definición de una clase
    def MiMetodo(self):                 # Definición de un método ('self' representa un objeto)
        self.MiPropiedad = "Mi Dato"    # Definición de una propiedad de objeto

MiObjeto = MiClase()                    # Instanciación de una clase
MiObjeto.MiMetodo()                     # Ejecución de un método
print MiObjeto.MiPropiedad              # Obtención de la propiedad de un objeto
```

Cada objeto es capaz de recibir mensajes, procesar datos, y enviar mensajes a otros objetos, como si se tratara de una “máquina” o caja negra, independiente, con un papel distinto o una responsabilidad única. Tanto los datos (llamados *propiedades*) como las acciones sobre esos datos (llamadas *métodos*) están estrechamente asociados con el objeto correspondiente.

Clase

Una clase es un modelo o plantilla que describe un tipo de objeto. El intérprete o compilador utiliza la clase cada vez que se tenga que crear un nuevo objeto. Este proceso se llama **instanciación**. Cada objeto es una instancia de alguna clase.

Objeto (Instancia)

Un objeto es una entidad de programación que contiene **propiedades** y **métodos**, de acuerdo a un modelo definido al que se denomina **clase**. Cada objeto tiene una **identidad** (nombre o identificador), un **estado** (datos o propiedades) y un **comportamiento** (acciones o métodos).

Propiedades

Las propiedades pueden ser de clase o de instancia:

```
class OtraClase:
    PropiedadClase = 0
    def __init__(self, n):          # Método constructor de la clase
        self.PropiedadObjeto = n

Objeto1 = OtraClase(1)              # Instancia 1: Inicializa la propiedad de Objeto1
Objeto2 = OtraClase(2)              # Instancia 2: Inicializa la propiedad de Objeto2
print Objeto1.PropiedadClase        # Imprime: 0
print Objeto1.PropiedadObjeto       # Imprime: 1
print Objeto2.PropiedadClase        # Imprime: 0
print Objeto2.PropiedadObjeto       # Imprime: 2
OtraClase.PropiedadClase = 3        # Modifica la propiedad de la clase
print Objeto1.PropiedadClase        # Imprime: 3
print Objeto2.PropiedadClase        # Imprime: 3
```

de Clase

Una propiedad de clase es aquella cuyo estado es compartido por todas las instancias de la clase.

de Instancia

Una propiedad de instancia es aquella cuyo estado puede ser invocado o modificado únicamente por la instancia a la que pertentece.

Métodos

Un método es una función, procedimiento o subrutina que forma parte de la definición de una clase. Los métodos, mediante sus parámetros de entrada, definen el comportamiento de las instancias de la clase asociada.

El parámetro "self"

La palabra `self` representa a la **instancia de una clase**, es decir, a un objeto. Se utiliza dentro de la clase para hacer referencia a las propiedades del objeto en sí y diferenciarlas de las de la propia clase.

```
class Persona:

    def __init__(self, nombre):
        self.nombre = nombre

    def saludar(self):
        print "Hola, mi nombre es", self.nombre

    def __del__(self):
        print "%s dice adiós." % self.nombre
```

```
carlos = Persona("Carlos Zayas")
carlos.saludar()
```

En el ejemplo, `nombre` es un **atributo** o **parámetro** del método `__init__`, mientras que `self.nombre` es una **propiedad de instancia**.

Sobrecarga de operadores

Consiste en la posibilidad de contar con métodos del mismo nombre pero con comportamientos diferentes.

```
class Lista:

    def __init__(self):
        self.items = []

    def __str__(self):
        return '\n'.join(self.items)

    def __repr__(self):
        return repr(self.items)

    def agregar(self, item):
        self.items.append(str(item))

    def vaciar(self):
        self.items = []

    def vacio(self):
        return self.items == []

    def cantidad(self):
        return len(self.items)

class Archivo:

    def __init__(self, nombre='lista.txt'):
        self.items = open(nombre).readlines() if os.path.exists(nombre) else []
        self.items = [ item.strip() for item in self.items ] # Limpia la lista
        self.nombre = nombre

    def __del__(self):
        open(self.nombre, 'w').write('\n'.join(self.items))

class ListArch(Archivo, Lista):

    pass

lista = ListArch()
lista.vaciar()
lista.agregar('Hola')
lista.agregar(2)
lista.agregar((3, 4))
lista.agregar('Chau')
print "Cantidad:", lista.cantidad(), '\n'
print lista
print '\nLista:',repr(lista)
```

En los ejemplos anteriores, `__init__` y `__str__` son **redefiniciones de métodos predefinidos** que modifican parte del comportamiento estándar de los objetos, en

este caso la construcción y la impresión del objeto, respectivamente. A esta capacidad del lenguaje se le denomina **sobrecarga**.

Constructor y Destructor

En Python, los métodos `__init__` y `__del__` son lo que se conoce como método **constructor** y **destructor** respectivamente. No es obligatorio incluirlos cuando definimos una clase, pero como se verá a continuación, pueden ser de mucha utilidad, y a menudo son necesarios.

El método **constructor** de una clase se ejecuta en cada instanciación de la clase, es decir, durante el proceso de creación de un objeto.

El método **destructor** de una clase se ejecuta cuando el objeto ya no es referenciado dentro del programa, o simplemente cuando el proceso termina.

Recordemos que un proceso es un programa en ejecución, y durante su finalización, se borran de la memoria todos los objetos creados, lo que activa el método **destructor** correspondiente a cada uno de ellos.

Getter / Setter / Deleter

Las propiedades, tanto las de clase como las de instancia, se pueden acceder directamente utilizando la sintaxis "punto" (`objeto.propiedad`). Sin embargo, se recomienda que cualquier operación relacionada con propiedades se realice mediante métodos especiales, denominados *getters* y *setters* (obtenedores y colocadores). Opcionalmente, también puede definirse un *deleter* (borrador).

Estos métodos especiales son necesarios principalmente para el manejo de las propiedades más importantes del objeto, generalmente aquellas que necesitan ser accedidas desde otros objetos, no precisamente desde los métodos propios del objeto.

No es obligatorio definir *getters* y *setters* para todas las propiedades, sino sólo para aquellas en las que necesitemos algún tipo de validación previa antes de obtener o colocar el valor involucrado.

En Python contamos con dos maneras de definir estos métodos especiales:

```
# Ejemplo 1: Función 'property'

class Ejemplo1(object):

    def __init__(self):
        self._x = None

    def getx(self):
        return self._x

    def setx(self, valor):
        self._x = valor
```

```
    def delx(self):
        del self._x

    x = property(getx, setx, delx, "Soy la propiedad 'x'.")

# Ejemplo 2: Decoradores ('@...')

class Ejemplo2(object):

    def __init__(self):
        self._x = None

    @property
     def x(self):
        "Soy la propiedad 'x'."
        return self._x

    @x.setter
     def x(self, valor):
        self._x = valor

    @x.deleter
     def x(self):
        del self._x
```

Ambas sintaxis cumplen el mismo propósito y dan exactamente los mismos resultados, pero la segunda es la más nueva incorporación al lenguaje Python.

También contamos con los métodos `__getitem__` y `__setitem__` para manipular listas y/o diccionarios usando únicamente el nombre del objeto. De esta manera, en vez de escribir `objeto.lista[indice]` simplemente podemos escribir `objeto[indice]`.

Ejemplo:

```
class Dic:

    def __init__(self):
        self.dic = {}

    def __repr__(self):
        return repr(self.dic)

    def __getitem__(self, clave):
        return self.dic.get(clave, None)

    def __setitem__(self, clave, valor):
        self.dic[clave] = valor
```

De esta manera, una instancia de la clase `Dic` puede usarse como si se tratase de un diccionario:

```
d = Dic()
d['Nombre'] = 'Carlos'
print d['Nombre']
```

Características Fundamentales

Encapsulamiento

Consiste en colocar al mismo nivel de abstracción a todos los elementos (estado y comportamiento) que pueden considerarse pertenecientes a una misma entidad (identidad). Esto permite aumentar la cohesión de los componentes del sistema.

```
class Persona(object):

    def __init__(self, nombre):
        self.setNombre(nombre)

    def getNombre(self):
        return ' '.join(self.__nombre)

    def setNombre(self, nombre):
        self.__nombre = nombre.split()

    nombre = property(getNombre, setNombre)
```

Las propiedades pertenecen al espacio de nombres del objeto (*namespace*) y pueden estar ocultas, es decir, sólo accesibles para el objeto. En Python, esto último se logra superficialmente anteponiendo dos guiones bajos (__) al nombre de la propiedad.

```
carlos = Persona("Carlos Zayas")
print carlos.nombre.upper()

carlos.nombre = 'Carlos A. Zayas G.'
print carlos.nombre.upper()
```

Decimos que en Python se logra sólo superficialmente el ocultamiento de las propiedades porque, si bien no podemos invocar la propiedad `nombre` de manera tradicional, sí podemos hacerlo de la siguiente manera:

```
print carlos._Persona__nombre
```

Herencia

Las clases se relacionan entre sí dentro de una jerarquía de clasificación que permite definir nuevas clases basadas en clases preexistentes, y así poder crear objetos especializados.

```
class Empleado(Persona):

    def __init__(self, nombre, cargo):
        Persona.__init__(self, nombre)
        self.cargo = cargo
        print "%s es %s" % (self.nombre, self.cargo)
```

La herencia es el mecanismo por excelencia de la programación orientada a objetos

que permite lograr la reutilización de código. Mediante ella, podemos crear nuevas clases modificando clases ya existentes.

Herencia Múltiple

Cuando la herencia involucra a más de una clase, hay herencia múltiple. El orden en el que las clases son invocadas determina la prevalencia de propiedades y métodos de idéntica denominación en el "árbol genealógico".

Orden de Resolución de Métodos

El Orden de Resolución de Métodos o MRO (*Method Resolution Order*) es la manera en que un lenguaje de programación decide dónde buscar un método (o una propiedad) en una clase que hereda estos elementos de varias clases superiores.

La importancia del MRO se hace patente en presencia de la herencia múltiple donde, ante dos elementos con la misma denominación en clases distintas, es necesario definir qué método o propiedad prevalecerá en una instanciación.

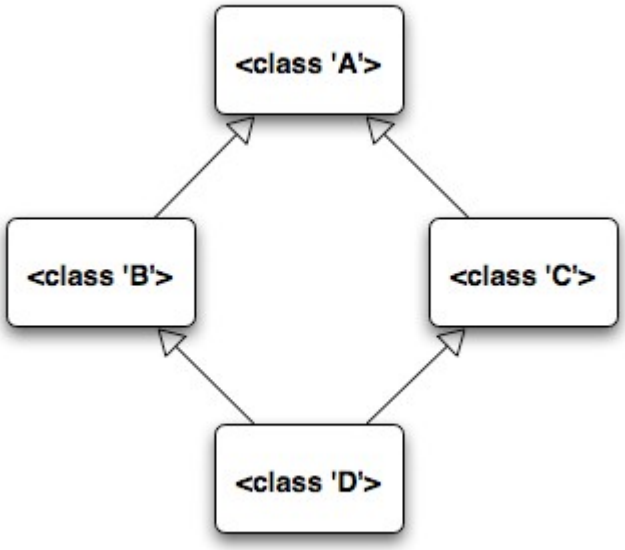

Figura 1: Diagrama de herencia "en diamante"

En el caso de Python, la evolución del lenguaje dio como resultado dos algoritmos MRO distintos, uno simple para las clases de estilo antiguo y otro más sofisticado para las clases de estilo nuevo (las que heredan de `object`). Ambos algoritmos provienen de la **teoría de grafos**, y son los siguientes:

DFS – *Depth First Search* (Búsqueda en profundidad): Recorre los nodos del grafo (árbol) de izquierda a derecha empezando de la raíz pero hasta el nodo más lejano. En el diagrama, empezando desde la clase D, el orden sería D, B, A, C.

BFS – *Breadth First Search* (Búsqueda en anchura): Recorre los nodos del grafo efectuando barridos de izquierda a derecha. En el diagrama, empezando desde la clase D, el orden sería D, B, C, A.

```
# Algoritmo: DFS - Depth First Search (Busqueda en profundidad)
class A: x = 'a'
class B(A): pass
class C(A): x = 'c'
class D(B, C): pass
print 'Viejo estilo: D.x = "%s"' % D.x

# Algoritmo: BFS - Breadth First Search (Busqueda en anchura)
class A(object): x = 'a'
class B(A): pass
class C(A): x = 'c'
class D(B, C): pass
print 'Nuevo estilo: D.x = "%s"' % D.x
```

```
Viejo estilo: D.x = "a"
Nuevo estilo: D.x = "c"
```

El MRO puede obtenerse mediante mecanismos de introspección, como puede verse en la siguiente secuencia de sentencias ejecutadas durante una sesión con el intérprete interactivo de Python:

Empecemos creando dos clases vacías, únicamente a modo de ejemplo, haciendo uso de la sentencia comodín `pass`:

```
>>> class animal(object): pass
...
>>> class perro(animal): pass
...
```

Acabamos de definir dos clases: `animal` (de tipo `object`) y `perro`, que hereda los métodos y propiedades de `animal`. Ambas clases aparentemente son iguales, a ninguna de las dos se les definieron métodos o propiedades específicas, pero se diferencian en cuanto a la herencia – una deriva de la otra.

A continuación, creamos una instancia de la clase `perro`, a la que llamamos `fido`.

```
>>> fido = perro()
```

Mediante el método `__class__` y el operador `is` podemos ver que la clase del objeto `fido` es `perro`, no `animal`, aunque "desciende" de ella.

```
>>> fido.__class__ is animal
False
>>> fido.__class__ is perro
True
```

Sin embargo, con la función `isinstance`, vemos que `fido` es una instancia de la clase `animal`, al igual que de la sub-clase `perro`.

```
>>> isinstance(fido, animal)
True
>>> isinstance(fido, perro)
True
```

Por último, el método `__mro__` nos devuelve una tupla en la que podemos ver el orden de resolución de métodos:

```
>>> animal.__mro__
(<class '__main__.animal'>, <type 'object'>)
>>> perro.__mro__
(<class '__main__.perro'>, <class '__main__.animal'>, <type 'object'>)
```

Los métodos y propiedades de una clase se superpondrán a los de la clase superior en el orden que aparecen en la tupla.

Polimorfismo

Consiste en definir comportamientos diferentes basados en una misma denominación, pero asociados a objetos distintos entre sí. Al llamarlos por ese nombre común, se utilizará el adecuado al objeto que se esté invocando.

```
a = Persona("Pablo")              # Nace Pablo
b = Persona("Juan")               # Nace Juan
print b                           # Juan dice "Hola"
c = Empleado("Esteban","Chofer")  # Nace Esteban
                                  # Esteban es Chofer
```

En el siguiente ejemplo, las dos clases derivadas de `Animal` comparten el método `sonido`, pero cada una le agrega su particularidad.

```
class Animal:

    cantidad = 0

    def __init__(self):
        print "Hola, soy un animal."
        self.nombre = ""
        Animal.cantidad += 1
        print "Hay", Animal.cantidad, "animales."

    def sonido(self):
        print "Este es mi sonido:"

    def __del__(self):
        print self.nombre, "dice: Adios!"

class Perro(Animal):

    def __init__(self, nombre):
        Animal.__init__(self)
        print "Soy un perro."
        self.nombre = nombre
        print "Me llamo", self.nombre

    def sonido(self):
        Animal.sonido(self)
```

```
        print "Guau!"

class Gato(Animal):

    def __init__(self, nombre):
        Animal.__init__(self)
        print "Soy un gato."
        self.nombre = nombre
        print "Me llamo", self.nombre

    def sonido(self):
        Animal.sonido(self)
        print "Miau!"

fido = Perro("Fido")
fido.sonido()
tom = Gato("Tom")
tom.sonido()
```

Los objetos `fido` y `tom` descienden de `Animal` pero cada uno “emite su propio sonido”.