Python

PARA TODOS

Raúl González Duque

Python

PARA TODOS

Raúl González Duque

Python para todos

por Raúl González Duque

Puede descargar la versión más reciente de este libro gratuitamente en la web http://mundogeek.net/tutorial-python/

La imágen de portada es una fotografía de una pitón verde de la especie Morelia viridis cuyo autor es Ian Chien. La fotografía está licenciada bajo Creative Commons Attribution ShareAlike 2.0

ORIENTACIÓN A OBJETOS

En el capítulo de introducción ya comentábamos que Python es un lenguaje multiparadigma en el se podía trabajar con programación estructurada, como veníamos haciendo hasta ahora, o con programación orientada a objetos o programación funcional.

La Programación Orientada a Objetos (POO u OOP según sus siglas en inglés) es un paradigma de programación en el que los conceptos del mundo real relevantes para nuestro problema se modelan a través de clases y objetos, y en el que nuestro programa consiste en una serie de interacciones entre estos objetos.

Clases y objetos

Para entender este paradigma primero tenemos que comprender qué es una clase y qué es un objeto. Un objeto es una entidad que agrupa un estado y una funcionalidad relacionadas. El estado del objeto se define a través de variables llamadas atributos, mientras que la funcionalidad se modela a través de funciones a las que se les conoce con el nombre de métodos del objeto.

Un ejemplo de objeto podría ser un coche, en el que tendríamos atributos como la marca, el número de puertas o el tipo de carburante y métodos como arrancar y parar. O bien cualquier otra combinación de atributos y métodos según lo que fuera relevante para nuestro programa.

Una clase, por otro lado, no es más que una plantilla genérica a partir

de la cuál instanciar los objetos; plantilla que es la que define qué atributos y métodos tendrán los objetos de esa clase.

Volviendo a nuestro ejemplo: en el mundo real existe un conjunto de objetos a los que llamamos coches y que tienen un conjunto de atributos comunes y un comportamiento común, esto es a lo que llamamos clase. Sin embargo, mi coche no es igual que el coche de mi vecino, y aunque pertenecen a la misma clase de objetos, son objetos distintos.

En Python las clases se definen mediante la palabra clave `class` seguida del nombre de la clase, dos puntos (:) y a continuación, indentado, el cuerpo de la clase. Como en el caso de las funciones, si la primera línea del cuerpo se trata de una cadena de texto, esta será la cadena de documentación de la clase o docstring.

```
class Coche:
    """Abstraccion de los objetos coche."""
    def __init__(self, gasolina):
        self.gasolina = gasolina
        print "Tenemos", gasolina, "litros"

    def arrancar(self):
        if self.gasolina > 0:
            print "Arranca"
        else:
            print "No arranca"

    def conducir(self):
        if self.gasolina > 0:
            self.gasolina -= 1
            print "Quedan", self.gasolina, "litros"
        else:
            print "No se mueve"
```

Lo primero que llama la atención en el ejemplo anterior es el nombre tan curioso que tiene el método `__init__`. Este nombre es una convención y no un capricho. El método `__init__`, con una doble barra baja al principio y final del nombre, se ejecuta justo después de crear un nuevo objeto a partir de la clase, proceso que se conoce con el nombre de instanciación. El método `__init__` sirve, como sugiere su nombre, para realizar cualquier proceso de inicialización que sea necesario.

Como vemos el primer parámetro de `__init__` y del resto de métodos

de la clase es siempre `self`. Esta es una idea inspirada en Modula-3 y sirve para referirse al objeto actual. Este mecanismo es necesario para poder acceder a los atributos y métodos del objeto diferenciando, por ejemplo, una variable local `mi_var` de un atributo del objeto `self.mi_var`.

Si volvemos al método `__init__` de nuestra clase `Coche` veremos cómo se utiliza `self` para asignar al atributo gasolina del objeto (`self.gasolina`) el valor que el programador especificó para el parámetro `gasolina`. El parámetro `gasolina` se destruye al final de la función, mientras que el atributo `gasolina` se conserva (y puede ser accedido) mientras el objeto viva.

Para crear un objeto se escribiría el nombre de la clase seguido de cualquier parámetro que sea necesario entre paréntesis. Estos parámetros son los que se pasarán al método `__init__`, que como decíamos es el método que se llama al instanciar la clase.

```
mi_coche = Coche(3)
```

Os preguntareis entonces cómo es posible que a la hora de crear nuestro primer objeto pasemos un solo parámetro a `__init__`, el número 3, cuando la definición de la función indica claramente que precisa de dos parámetros (`self` y `gasolina`). Esto es así porque Python pasa el primer argumento (la referencia al objeto que se crea) automágicamente.

Ahora que ya hemos creado nuestro objeto, podemos acceder a sus atributos y métodos mediante la sintaxis `objeto.atributo` y `objeto.metodo()`:

```
>>> print mi_coche.gasolina
3
>>> mi_coche.arrancar()
Arranca
>>> mi_coche.conducir()
Quedan 2 litros
>>> mi_coche.conducir()
Quedan 1 litros
>>> mi_coche.conducir()
Quedan 0 litros
>>> mi_coche.conducir()
```

```
No se mueve
>>> mi_coche.arrancar()
No arranca
>>> print mi_coche.gasolina
0
```

Como último apunte recordar que en Python, como ya se comentó en repetidas ocasiones anteriormente, todo son objetos. Las cadenas, por ejemplo, tienen métodos como `upper()`, que devuelve el texto en mayúsculas o `count(sub)`, que devuelve el número de veces que se encontró la cadena `sub` en el texto.

Herencia

Hay tres conceptos que son básicos para cualquier lenguaje de programación orientado a objetos: el encapsulamiento, la herencia y el polimorfismo.

En un lenguaje orientado a objetos cuando hacemos que una clase (subclase) herede de otra clase (superclase) estamos haciendo que la subclase contenga todos los atributos y métodos que tenía la superclase. No obstante al acto de heredar de una clase también se le llama a menudo "extender una clase".

Supongamos que queremos modelar los instrumentos musicales de una banda, tendremos entonces una clase `Guitarra`, una clase `Batería`, una clase `Bajo`, etc. Cada una de estas clases tendrá una serie de atributos y métodos, pero ocurre que, por el mero hecho de ser instrumentos musicales, estas clases compartirán muchos de sus atributos y métodos; un ejemplo sería el método `tocar()`.

Es más sencillo crear un tipo de objeto `Instrumento` con las atributos y métodos comunes e indicar al programa que `Guitarra`, `Batería` y `Bajo` son tipos de instrumentos, haciendo que hereden de `Instrumento`.

Para indicar que una clase hereda de otra se coloca el nombre de la clase de la que se hereda entre paréntesis después del nombre de la clase:

```
class Instrumento:
    def __init__(self, precio):
```

```
            self.precio = precio

        def tocar(self):
            print "Estamos tocando musica"

        def romper(self):
            print "Eso lo pagas tu"
            print "Son", self.precio, "$$$"

    class Bateria(Instrumento):
        pass

    class Guitarra(Instrumento):
        pass
```

Como `Bateria` y `Guitarra` heredan de Instrumento, ambos tienen un método `tocar()` y un método `romper()`, y se inicializan pasando un parámetro `precio`. Pero, ¿qué ocurriría si quisiéramos especificar un nuevo parámetro `tipo_cuerda` a la hora de crear un objeto `Guitarra`? Bastaría con escribir un nuevo método `__init__` para la clase `Guitarra` que se ejecutaría en lugar del `__init__` de Instrumento. Esto es lo que se conoce como sobreescribir métodos.

Ahora bien, puede ocurrir en algunos casos que necesitemos sobreescribir un método de la clase padre, pero que en ese método queramos ejecutar el método de la clase padre porque nuestro nuevo método no necesite más que ejecutar un par de nuevas instrucciones extra. En ese caso usaríamos la sintaxis `SuperClase.metodo(self, args)` para llamar al método de igual nombre de la clase padre. Por ejemplo, para llamar al método `__init__` de `Instrumento` desde `Guitarra` usaríamos `Instrumento.__init__(self, precio)`

Observad que en este caso si es necesario especificar el parámetro `self`.

Herencia múltiple

En Python, a diferencia de otros lenguajes como Java o C#, se permite la herencia múltiple, es decir, una clase puede heredar de varias clases a la vez. Por ejemplo, podríamos tener una clase `Cocodrilo` que heredara de la clase `Terrestre`, con métodos como `caminar()` y atributos como `velocidad_caminar` y de la clase `Acuatico`, con métodos como `nadar()` y atributos como `velocidad_nadar`. Basta con enumerar las clases de

las que se hereda separándolas por comas:

```
class Cocodrilo(Terrestre, Acuatico):
    pass
```

En el caso de que alguna de las clases padre tuvieran métodos con el mismo nombre y número de parámetros las clases sobreescribirían la implementación de los métodos de las clases más a su derecha en la definición.

En el siguiente ejemplo, como `Terrestre` se encuentra más a la izquierda, sería la definición de `desplazar` de esta clase la que prevalecería, y por lo tanto si llamamos al método `desplazar` de un objeto de tipo `Cocodrilo` lo que se imprimiría sería "El animal anda".

```
class Terrestre:
    def desplazar(self):
        print "El animal anda"

class Acuatico:
    def desplazar(self):
        print "El animal nada"

class Cocodrilo(Terrestre, Acuatico):
    pass

c = Cocodrilo()
c.desplazar()
```

Polimorfismo

La palabra polimorfismo, del griego *poly morphos* (varias formas), se refiere a la habilidad de objetos de distintas clases de responder al mismo mensaje. Esto se puede conseguir a través de la herencia: un objeto de una clase derivada es al mismo tiempo un objeto de la clase padre, de forma que allí donde se requiere un objeto de la clase padre también se puede utilizar uno de la clase hija.

Python, al ser de tipado dinámico, no impone restricciones a los tipos que se le pueden pasar a una función, por ejemplo, más allá de que el objeto se comporte como se espera: si se va a llamar a un método `f()` del objeto pasado como parámetro, por ejemplo, evidentemente el objeto tendrá que contar con ese método. Por ese motivo, a diferencia

de lenguajes de tipado estático como Java o C++, el polimorfismo en Python no es de gran importancia.

En ocasiones también se utiliza el término polimorfismo para referirse a la sobrecarga de métodos, término que se define como la capacidad del lenguaje de determinar qué método ejecutar de entre varios métodos con igual nombre según el tipo o número de los parámetros que se le pasa. En Python no existe sobrecarga de métodos (el último método sobreescribiría la implementación de los anteriores), aunque se puede conseguir un comportamiento similar recurriendo a funciones con valores por defecto para los parámetros o a la sintaxis `*params` o `**params` explicada en el capítulo sobre las funciones en Python, o bien usando decoradores (mecanismo que veremos más adelante).

Encapsulación

La encapsulación se refiere a impedir el acceso a determinados métodos y atributos de los objetos estableciendo así qué puede utilizarse desde fuera de la clase.

Esto se consigue en otros lenguajes de programación como Java utilizando modificadores de acceso que definen si cualquiera puede acceder a esa función o variable (`public`) o si está restringido el acceso a la propia clase (`private`).

En Python no existen los modificadores de acceso, y lo que se suele hacer es que el acceso a una variable o función viene determinado por su nombre: si el nombre comienza con dos guiones bajos (y no termina también con dos guiones bajos) se trata de una variable o función privada, en caso contrario es pública. Los métodos cuyo nombre comienza y termina con dos guiones bajos son métodos especiales que Python llama automáticamente bajo ciertas circunstancias, como veremos al final del capítulo.

En el siguiente ejemplo sólo se imprimirá la cadena correspondiente al método `publico()`, mientras que al intentar llamar al método `__privado()` Python lanzará una excepción quejándose de que no existe (evidentemente existe, pero no lo podemos ver porque es privado).

```
class Ejemplo:
    def publico(self):
        print "Publico"

    def __privado(self):
        print "Privado"

ej = Ejemplo()
ej.publico()
ej.__privado()
```

Este mecanismo se basa en que los nombres que comienzan con un doble guión bajo se renombran para incluir el nombre de la clase (característica que se conoce con el nombre de *name mangling*). Esto implica que el método o atributo no es realmente privado, y podemos acceder a él mediante una pequeña trampa:

```
ej._Ejemplo__privado()
```

En ocasiones también puede suceder que queramos permitir el acceso a algún atributo de nuestro objeto, pero que este se produzca de forma controlada. Para esto podemos escribir métodos cuyo único cometido sea este, métodos que normalmente, por convención, tienen nombres como `getVariable` y `setVariable`; de ahí que se conozcan también con el nombre de *getters* y *setters*.

```
class Fecha():
    def __init__(self):
        self.__dia = 1

    def getDia(self):
        return self.__dia

    def setDia(self, dia):
        if dia > 0 and dia < 31:
            self.__dia = dia
        else:
            print "Error"

mi_fecha = Fecha()
mi_fecha.setDia(33)
```

Esto se podría simplificar mediante propiedades, que abstraen al usuario del hecho de que se está utilizando métodos entre bambalinas para obtener y modificar los valores del atributo:

```
class Fecha(object):
    def __init__(self):
        self.__dia = 1

    def getDia(self):
        return self.__dia

    def setDia(self, dia):
        if dia > 0 and dia < 31:
            self.__dia = dia
        else:
            print "Error"

    dia = property(getDia, setDia)

mi_fecha = Fecha()
mi_fecha.dia = 33
```

Clases de "nuevo-estilo"

En el ejemplo anterior os habrá llamado la atención el hecho de que la clase `Fecha` derive de `object`. La razón de esto es que para poder usar propiedades la clase tiene que ser de "nuevo-estilo", clases enriquecidas introducidas en Python 2.2 que serán el estándar en Python 3.0 pero que aún conviven con las clases "clásicas" por razones de retrocompatibilidad. Además de las propiedades las clases de nuevo estilo añaden otras funcionalidades como descriptores o métodos estáticos.

Para que una clase sea de nuevo estilo es necesario, por ahora, que extienda una clase de nuevo-estilo. En el caso de que no sea necesario heredar el comportamiento o el estado de ninguna clase, como en nuestro ejemplo anterior, se puede heredar de `object`, que es un objeto vacio que sirve como base para todas las clases de nuevo estilo.

La diferencia principal entre las clases antiguas y las de nuevo estilo consiste en que a la hora de crear una nueva clase anteriormente no se definía realmente un nuevo tipo, sino que todos los objetos creados a partir de clases, fueran estas las clases que fueran, eran de tipo `instance`.

Métodos especiales

Ya vimos al principio del artículo el uso del método `__init__`. Existen otros métodos con significados especiales, cuyos nombres siempre comienzan y terminan con dos guiones bajos. A continuación se listan algunos especialmente útiles.

`__init__(self, args)`
Método llamado después de crear el objeto para realizar tareas de inicialización.

`__new__(cls, args)`
Método exclusivo de las clases de nuevo estilo que se ejecuta antes que `__init__` y que se encarga de construir y devolver el objeto en sí. Es equivalente a los constructores de C++ o Java. Se trata de un método estático, es decir, que existe con independencia de las instancias de la clase: es un método de clase, no de objeto, y por lo tanto el primer parámetro no es `self`, sino la propia clase: `cls`.

`__del__(self)`
Método llamado cuando el objeto va a ser borrado. También llamado destructor, se utiliza para realizar tareas de limpieza.

`__str__(self)`
Método llamado para crear una cadena de texto que represente a nuestro objeto. Se utiliza cuando usamos `print` para mostrar nuestro objeto o cuando usamos la función `str(obj)` para crear una cadena a partir de nuestro objeto.

`__cmp__(self, otro)`
Método llamado cuando se utilizan los operadores de comparación para comprobar si nuestro objeto es menor, mayor o igual al objeto pasado como parámetro. Debe devolver un número negativo si nuestro objeto es menor, cero si son iguales, y un número positivo si nuestro objeto es mayor. Si este método no está definido y se intenta comparar el objeto mediante los operadores <, <=, > o >= se lanzará una excepción. Si se utilizan los operadores == o != para comprobar si dos objetos son iguales, se comprueba si son el mismo objeto (si tienen el mismo id).

`__len__(self)`
Método llamado para comprobar la longitud del objeto. Se utiliza, por

ejemplo, cuando se llama a la función `len(obj)` sobre nuestro objeto. Como es de suponer, el método debe devolver la longitud del objeto.

Existen bastantes más métodos especiales, que permite entre otras cosas utilizar el mecanismo de slicing sobre nuestro objeto, utilizar los operadores aritméticos o usar la sintaxis de diccionarios, pero un estudio exhaustivo de todos los métodos queda fuera del propósito del capítulo.