Introducción a la programación orientada a objetos con Python

Notas acerca de este artículo

Este artículo ha sido extraído de la página web http://blog.rvburke.com cumpliendo con la norma de copyright establecida.

Su autor original es Rafael Villar Burke y el artículo data del 22 de Noviembre de 2006. La versión en PDF ha sido realizada por Oscar Carballal Prego y data del 24 de Noviembre de 2009.

Se ha procedido a hacer algunas concesiones visuales de cara a orientar al lector de forma más intuitiva a lo que está viendo en cada momento. Los textos contenidos en cajas grises se refieren a líneas de código, y los textos contenidos en cajas con borde negro son las representaciones de los datos que deberían salir por pantalla. Esto se junta con las concesiones ya establecidas en el artículo original, las cuales son mostradas más abajo.

"I code, therefore I am."

Este artículo es una introducción a la programación orientada a objetos (POO, o OOP por sus siglas en inglés), uno de los paradigmas de programación estructurada más importante hoy en día.

El artículo usa el lenguaje **Python** para los distintos ejemplos, y explica la implementación de la POO en ese lenguaje, aunque aborda conceptos generales que son fácilmente trasladables a otros lenguajes.

Los fragmentos de código se señalan usando un estilo de texto diferente (ancho fijo) y, cuando se utiliza el intérprete de python, se prefija cada línea con los símbolos '>>>'. Esta introducción presupone conocimientos básicos de programación y del lenguaje python.

Formas de pensar: paradigmas de programación

Uno de los elementos básicos a la hora de realizar un programa es la modelización del problema que pretende resolver. Es preciso localizar las variables y los aspectos relevantes, así como comprender los pasos necesarios para obtener el resultado a partir de los datos iniciales, etc. Es decir, abstraer el problema, reduciendo sus detalles de forma que podamos trabajar con pocos elementos cada vez.

La algoritmia plantea la forma óptima de resolver problemas concretos, tales como la ordenación de una lista de elementos, la búsqueda de un elemento en un conjunto, la manipulación de conjuntos de datos, etc. Sin embargo, no proporcionan un marco general, un enfoque, que nos permita plantear y formular las soluciones a un problema de forma coherente.

Los paradigmas de programación llenan ese hueco, proporcionando guías tanto sobre cómo realizar la abstracción de los datos como sobre el control de la ejecución. Es decir, **los paradigmas de programación son herramientas conceptuales para analizar, representar y abordar los problemas, presentando sistematizaciones alternativas o complementarias para pasar del espacio de los problemas al de las implementaciones de una solución**.

Es muy recomendable y productivo comprender el enfoque de distintos paradigmas, puesto que presentan estrategias alternativas, incrementando nuestras herramientas disponibles y haciéndono reflexionar sobre muchas de las tareas que realizamos al crear un programa. Además, a menudo ocurre que unos problemas se formulan de forma muy clara si se los analiza según una perspectiva determinada, mientras que producen una gran complejidad vistos de otra manera.

Algunos paradigmas habituales

Algunas de las formas de pensar los problemas que ha llegado a sistematizarse como paradigmas de programación son:

La **programación modular**: Los programas se forman por partes separadas llamadas módulos. Estos funcionan de forma independiente entre sí, o se relacionan a través de interfaces bien definidas.

- La **programación procedural**: Un programa se compone de procedimientos (subrutinas, métodos o funciones) que son fragmentos de código que pueden llamarse desde cualquier punto de la ejecución de un programa, incluídos otros procedimientos o él mismo. (ej. ALGOL)
- La **programación estructurada**: Se puede expresar cualquier programa utilizando únicamente tres estructuras de control: secuencial, condicional e iterativa. Los programas se componen de partes menores con un único punto de entrada, y se trata de aislarlos para evitar la complejidad que introducen los efectos colaterales. (ej. Pascal, C, Ada)
- La **programación imperativa**: Un programa se puede definir en términos de estado, y de instrucciones secuenciales que modifican dicho estado. (ej. C, BASIC)
- La **programación declarativa**: Es posible expresar un programa a través de condiciones, proposiciones o restricciones, a partir de los que se obtiene la solución mediante reglas internas de control. (ej. PROLOG)
- La **programación funcional**: Se puede expresar un programa como una secuencia de aplicación de funciones. Elude el concepto de estado del cómputo y no precisa de las estructuras de control de la programación estructurada. (ej. LISP, Haskell)
- La **programación orientada a objetos**: Los programas se definen en términos de "clases de objetos" que se comunican entre sí mediante el envío de mensajes. Es una evolución de los paradigmas de la programación procedural, estructurada y modular, y se implementa en lenguajes como Java, Smalltalk, Python o C++.

Programación multiparadigma

Los paradigmas de programación son idealizaciones, y, como tales, no siempre se presentan de forma totalmente 'pura', ni siempre resultan incompatibles entre sí. Cuando se entremezclan diversos paradigmas se produce lo que se conoce como **programación multiparadigma**.

El uso de distintos modelos, según se adapten mejor a las diversas partes de un problema o a nuestra forma de pensamiento, resulta también más natural y permite expresar de forma más clara y concisa nuestras ideas.

Algunos lenguajes, como **Haskell** o **Prolog** están diseñados para encajar perfectamente en la visión de un paradigma particular, mientras que otros, como **python**, se adaptan especialmente bien a la programación multiparadigma y dan gran libertad a la hora de resolver un problema, al no imponen un mismo patrón para todos los casos.

Tipos de datos

Una forma de almacenar y representar una fecha en un programa podría ser la siguiente:

```
d = 14
m = "Noviembre"
```

```
a = 2006
def dime_fecha(dia, mes, anho):
    return "%i de %s de %i del calendario gregoriano" % (dia, mes, anho)
print fecha(d, m, a)
```

para obtener la siguiente salida:

```
"14 de Noviembre de 2006"
```

En este ejemplo, para representar y manipular lo que conceptualmente entendemos como una fecha, se utilizan las variables d, m y a como almacenes de datos, y un procedimiento o función, de nombre dime_fecha, para realizar la representación de la fecha almacenada.

Las variables permiten almacenar tipos de datos concretos como los enteros (d y a) o las cadenas de texto (m), y la función realiza su trabajo devolviendo otro tipo concreto, una cadena de texto.

Uno de los problemas de este enfoque es que no refleja demasiado bien el concepto fecha que usamos como modelo mental al implementarlo como un conjunto de tres variables no relacionadas entre sí (d, m y a). Es decir, las tres variables tienen una unidad conceptual en nuestra mente, que no se refleja en la forma de expresar el problema. De la misma manera, la función dime_fecha desconoce igualmente la relación entre sus parámetros, que podría ser totalmente arbitraria.

Esta situación, que puede parecer una cuestión de 'estilo', implica un riesgo de problemas de sincronización entre variables 'relacionadas', dispersión en distintas partes del código de manipulaciones que son relevantes al conjunto, pérdida del valor semántico de cada variable que puede dar lugar a errores al transformar el código…

Un ejemplo de desconexión semántica entre la implementación y el concepto es que d puede ser cualquier valor entero, mientras que un día tiene un valor situado entre 1 y 31, m es una cadena de texto dentro de un número limitado de opciones, o a no puede tomar valores negativos. Por supuesto que esto se podría solucionar con código que hiciese comprobaciones adicionales, pero en este caso queremos resaltar cómo un programa es un modelo aproximado a un problema y cómo sería preferible que la implementación pudiese representar de forma más ajustada el comportamiento del modelo elegido.

Una posible solución sería disponer de un tipo de datos que represente de forma más adecuada el concepto de fecha y que nos permita manipular fechas de la misma manera que manipulamos números enteros, números reales o cadenas de texto, independientemente de la implementación interna que los sustente.

Clases y objetos

Al enunciar algunos tipos de paradigmas hemos visto que la **programación orientada a objetos define los programas en términos de "clases de objetos" que se comunican entre sí mediante el envío de mensajes**. En este apartado aclararemos qué se entiende en ese contexto por clases y objetos.

Las clases surgen de la generalización de los tipos de datos y **permiten una representación más directa de los conceptos necesarios para la modelización de un problema permitiendo definir nuevos tipos al usuario**.

Las clases permiten agrupar en un nuevo tipo los datos y las funcionalidades asociadas a dichos datos, favoreciendo la separación entre los detalles de la implementación de las propiedades esenciales para su uso. A

esta cualidad, de no mostrar más que la información relevante, ocultando el estado y los métodos internos de la clase, es conocida como **"encapsulación"**, y es un principio heredado de la programación modular.

Un aspecto importante en el uso de clases es que no se manipulan directamente (salvo en lo que se conoce como *metaprogramación*), sino que sirven para la definición nuevos tipos. **Una clase define propiedades y comportamiento que se muestran en los entes llamados *objetos*** (o **instancias de una clase**). La clase actúa como molde de un conjunto de objetos, de los que se dice que **pertenecen** a la clase.

Trasladando estos conceptos a los números enteros (tipo *int*), se puede decir que las variables que almacenan números enteros son objetos o instancias de la clase *int* o que *pertenecen* a la clase *int*.

En términos más abstractos, podemos pensar en las clases como definiciones y en los objetos como expresiones concretas de dichas definiciones.

Fisonomía de una clase

En python, el esquema para la definición de una clase es el siguiente:

```
class NombreClase:
  <instrucción_1>
  ...
  <instrucción_n>
```

en donde **class** es una palabra reservada que indica la declaración de una clase, **NombreClase** una etiqueta que da nombre a la clase y, los dos puntos, que señalan el inicio del bloque de instrucciones de la clase.

El cuerpo de instrucciones de la clase puede contener tanto asignaciones de datos como definiciones de funciones. Este bloque de instrucciones se encuentra en el nuevo espacio de nombres con el nombre de la clase, que se genera, a su vez, con la creación de cada objeto. En ambos casos, en el espacio de nombres de la clase y del objeto, es posible acceder a los elementos que lo integran usando el operador punto, como en el caso de los espacios de nombres de un módulo (e.j. *math.pi*, *objeto.dato*, *objeto.funcion()*).

Si la primera instrucción del cuerpo de la clase es una cadena de texto, ésta se usa como cadena de documentación de la clase.

Nuestro ejemplo podría reescribirse en términos de una clase así:

```
class Fecha:
    "Ejemplo de clase para representar fechas"
    dia = 14
    mes = "Noviembre"
    anho = 2006
    def dime_fecha(self):
        return "%i de %s de %i" % (Fecha.dia, Fecha.mes, Fecha.anho)

mi_fecha = Fecha()
print mi_fecha.dia, mi_fecha.mes, mi_fecha.anho
print mi_fecha.dime_fecha()
```

En el fragmento de código anterior definimos la clase Fecha y ella asignamos valores a las etiquetas dia, mes y anho (atributos de la clase), y definimos una función (un método de la clase), dime_fecha.

Fuera de la definición de la clase creamos (instanciamos) un objeto perteneciente a la clase Fecha. Esta instanciación se realiza utilizando la notación de llamada a función, y podemos entenderla como una llamada a una función que devuelve un objeto de la clase Fecha. El objeto obtenido es asignado a la etiqueta mi_fecha.

En la siguientes instrucciones mostramos por pantalla los valores de las propiedades dia, mes y anho y el resultado de la llamada al método dime_fecha, usando el operador punto para acceder a las propiedades y métodos del objeto mi_fecha, ya que, como hemos mencionado antes, forman parte del espacio de nombres del objeto.

La llamada al método dime_fecha no incluye ningún parámetro, pero podemos comprobar que, de forma contradictoria, la declaración del método indica uno, self. Este parámetro es añadido automáticamente por el intérprete en las llamadas a los métodos de una clase, y contiene una referencia al objeto que recibe la señal (el que recibe una llamada a un método). self permite acceder a las propiedades y métodos de un objeto concreto y aclararemos su significado y uso más adelante. Por ahora nos basta saber que **es necesario incluir** self **como primer parámetro de los métodos definidos en una clase** y que **no es preciso incluir el parámetro implícito** self **al realizar llamadas a un método a través del objeto al que pertenece**.

En realidad, en el ejemplo anterior, la llamada mi_fecha.dime_fecha() es una forma más cómoda de escribir (azúcar sintáctico) Fecha.dime_fecha(mi_fecha), lo que explica la presencia del parámetro.

Un paseo entre objetos

Hasta el momento hemos visto cómo las clases definen los datos (o estado) y comportamiento de los objetos a través de atributos (o propiedades) y métodos.

Ahora vamos a hacer un recorrido, utilizando una sesión interactiva del intérprete y las capacidades de introspección de python, para explicar el uso y el comportamiento de los objetos.

Veremos algunas **características de los objetos**:

- **Identidad**. Los objetos se diferencian entre sí, de forma que dos objetos, creados a partir de la misma clase y con los mismos parámetros de inicialización, son entes distintos.
- Definen su **comportamiento** (y operar sobre sus datos) a través de **métodos**, equivalentes a funciones.
- Definen o reflejan su **estado** (datos) a través de **propiedades o atributos**, que pueden ser tipos concretos u otros objetos.

Pasamos a ver una sesión interactiva del intérprete:

```
>>> mi_fecha = Fecha()
```

```
>>> mi_fecha_2 = Fecha()
>>> mi_fecha is mi_fecha_2
False
>>> print mi_fecha
<__main __.Fecha instance at 0x00B184B8>
>>> print mi_fecha_2
<__main __.Fecha instance at 0x00B18328>
```

Vemos que tanto mi_fecha como mi_fecha_2 son instancias de la clase Fecha (en el módulo __main__) y cada una tiene su propio espacio de memoria independiente.

```
 >>> print mi fecha.dime fecha()
14 de Noviembre de 2006
>>> print mi fecha.dia
14
>>> print mi fecha 2.dia
14
>>> Fecha.dia = 16
>>> print Fecha.dia
16
>>> print mi fecha.dia
16
>> print mi fecha 2.dia
16
>>> mi fecha.dia = 30
>>> print Fecha.dia
16
>>> print mi fecha.dia
30
>>> print mi fecha 2.dia
16
```

Este último experimento observamos cómo, pese a que mi_fecha y mi_fecha_2 son dos instancias distintas de la clase Fecha, ambas hacen referencia a través de su atributo dia al mismo valor de la clase Fecha.dia. Sin embargo, cuando se produce una asignación a ese atributo (a esa etiqueta, en realidad) en uno de los objetos vemos cómo, a partir de entonces, ese objeto dispone de un valor 'individual' asociado al atributo.

Ese comportamiento revela la existencia de **atributos (o estado) compartidos por todas las instancias de una clase** y de **atributos (o estado) ligados a una instancia particular**. Los primeros se sitúan en el espacio de nombres de la clase, y vemos cómo se pueden hacer asignaciones en la sesión interactiva anterior, o lecturas, en el método dime_fecha del ejemplo anterior. Los segundos se sitúan en el espacio de nombres de cada objeto, y ahí es donde resulta útil el parámetro self.

Antes de proseguir, podemos escudriñar qué símbolos son accesibles y qué contienen los espacios de nombres de la clase Fecha y de los objetos mi_fecha y mi_fecha_2. Usamos la función *dir* para lo primero, y el atributo __dict__ para lo segundo, ya que en python los espacios de nombres se implementan como *diccionarios*:

```
 >>> dir(Fecha)
['__doc__', '__module__', 'anho', 'dia', 'dime_fecha', 'mes']
>>> dir(fecha3)
['__doc__', '__module__', 'anho', 'dia', 'dime_fecha', 'mes']
>>> dir(fecha4)
['__doc__', '__module__', 'anho', 'dia', 'dime_fecha', 'mes']
```

En los tres podemos acceder a los mismos símbolos. Sin embargo...

```
>>> Fecha.__dict__
{'__module__': '__main__', 'anho': 2006, 'dime_fecha':  x00B176F0>, 'dia':
14, 'mes': 'Noviembre', '__doc__': 'Ejemplo de clase para representar
fechas'}
>>> mi_fecha_2.__dict__
{}
>>> mi_fecha.__dict__
{'dia': 30}
```

El espacio de nombres de la clase es el que alberga los nombres de los atributos y métodos indicados en la declaración de la clase, además de los atributos especiales __doc__, que contiene la cadena de documentación y

__module__, que contiene el nombre del módulo al que pertenece la clase. Esos símbolos son accesibles también para los objetos pertenecientes a la clase (tal como nos indicaba *dir()*).

Podemos ver también que el espacio de nombres del objeto mi_fecha_2 se encuentra vacío, mientras que el del objeto mi_fecha contiene el atributo día. Esto se debe a que anteriormente realizamos una asignación al atributo dia del objeto mi_fecha (dándole el valor 30). El espacio de nombres recoge esa versión particular del atributo, que no aparece en el otro objeto, puesto que accede al atributo de clase.

A los atributos compartidos por todos los objetos de una clase se los denomina **atributos de clase**, cuando se desea diferenciarlos de los atributos que pertenecen a cada instancia en particular.

Acceso individualizado a objetos: self

Ahora que hemos visto cómo usar atributos de clase para compartir datos entre todos los objetos de una misma clase, también nos interesa conocer cómo utilizar atributos comunes a una clase pero que puedan tomar valores distintos para cada uno de los objetos.

Retomamos ahora el misterioso parámetro self que vimos estaba presente como primer parámetro de todos los métodos de una clase. En su momento ya comentamos que self **contiene una referencia al objeto que recibe la señal** (el objeto cuyo método es llamado), por lo que **podemos usar esa referencia para acceder al espacio de nombres del objeto**, es decir, a sus atributos individuales.

```
>>> class F:
...     i = 5
...     def dime_i(self):
...             return F.i
...     def dime_mi_i(self):
...             return self.i
...
>>> a = F()
>>> a.dime_i()
5
>>> a.dime_mi_i()
5
>>> a.i = 6
>>> a.dime_i()
5
>>> a.dime_mi_i()
6
>>> a.i
6
```

En este ejemplo el método dime_i usa el atributo de clase y retorna su valor, 5, mientras que el método dime_mi_i usa una referencia al objeto y devuelve el valor 6, el asignado al atributo del objeto a. Así accedemos selectivamente al atributo de clase i o al atributo de instancia i.

Algo que puede aclarar algo más la naturaleza del parámetro self, es recordar que, cuando hacemos una llamada a un método un_metodo de un objeto un_objeto perteneciente a la clase UnaClase, el intérprete traduce la expresión un_objeto.un_metodo() como UnaClase.un_metodo(un_objeto).

El uso de self responde a la forma particular en que python implementa el soporte de objetos, aunque lenguajes

como Java o C++ utilizan mecanismos similares con la palabra reservada this. En python self no es una palabra reservada y cualquier etiqueta usada como primer parámetro valdría igualmente, aunque se desaconseja totalmente el uso de otras etiquetas, por tratarse de una convención muy arraigada en el lenguaje que hace el código más legible, facilita el resaltado de sintaxis, etc.

El método de inicialización __init__

Hemos visto ya cómo definir y usar atributos de clase y de instancia. Pero en python no es necesaria la declaración de variables, por lo que cualquier atributo al que se realice una asignación en el código se convierte en un atributo de instancia:

```
>>> a = F()
>>> a.i = 6
>>> a.atr = 3
>>> print a.atr
3
>>> a.__dict__
{'i': 6, 'atr': 3}
```

Una funcionalidad muy conveniente que hemos visto en la declaración de atributos de clase es la de realizar su inicialización en el cuerpo de la clase. Para poder inicializar los atributos de una instancia existe un método *especial* que permite actuar sobre la inicialización del objeto. Dicho método se denomina __init__ (con dos guiones bajos al principio y al final) y admite cualquier número de parámetros, siendo el primero la referencia al objeto que es incializado (self, por convención), que nos permite realizar las asignaciones a atributos de la instancia. Este método se ejecuta siempre que se crea un nuevo objeto de la clase, tras la asignación de memoria y con los atributos de clase inicializados, y se corresponde parcialmente con el concepto de **constructor de la clase** existente en otros lenguajes.

Con la referencia que nos proporciona self podemos inicializar atributos de instancia, con valores predeterminados si lo deseamos, como en cualquier definición de función:

```
class Clase:
    def __init__(self, x=2):
        self.x = x
        self.a = x**2
        self.b = x**3
        self.c = 999
    def dime_datos(self):
        return "Con x=%i obtenemos: a=%i, b=%i, c=%i" % (self.x, self.a,
self.b, self.c)
a1 = Clase(2)
a1.dime_datos()
a2 = Clase(3)
a2.dime_datos()
```

Salida:

```
Con x=2 obtenemos: a=4, b=8, c=999 Con x=3 obtenemos: a=9, b=27, c=999
```

Propiedades y Atributos

Aunque en la terminología general de la programación orientada a objetos *atributo y propiedad* se pueden utilizar como sinómimos, en python se particulariza el uso de **propiedades** para un tipo especial de **atributos**

cuyo acceso se produce a través de llamadas a funciones.

```
class ClaseC(object):
    def __init__(self):
        self.__b = 0
    def __get_b(self):
        return self.__b
    def __set_b(self, valor):
        if valor > 10:
            self.__b = 0
        else:
            self.__b = valor
    b = property(__get_b, __set_b, 'Propiedad b')

c1 = ClaseC()
print c1.b
# b es 0
c1.b = 5
print c1.b
# b es 5
c2 = ClaseC()
print c2.b
# b es 0
c2.b = 12
print c2.b
#b es 0
```

En este ejemplo se define la propiedad b, cuyo comportamiento es: cuando se realiza una asignación, toma el valor entregado si es menor que *10*, o *0* en caso contrario, y lo almacena en un atributo privado __b. Cuando se produce la lectura de b, devuelve el valor guardado en el atributo privado.

En Python, para poder usar propiedades en una clase es necesario hacerla derivar de la clase object de ahí que aparezca al lado del nombre de la clase ClaseC. En un apartado posterior se explicará en qué consiste la derivación de clases.

La signatura de la función property, que define una propiedad de la clase es la siguiente: nombre_propiedad = property(get_f, set_f, del_f, doc) donde get_f es la función llamada cuando se produce la lectura del atributo; set_f la función llamada cuando se produce una asignación al atributo; del_f la función llamada cuando se elimina el atributo, y doc es una cadena de documentación de la propiedad.

Las propiedades resultan muy útiles para realizar la validación de los valores asignados, la transformación o cálculo de valores devueltos y, además, dotan de gran flexibilidad a la hora de escribir el código, puesto que es posible empezar usando atributos y luego sustituirlo por una propiedad que realice funciones adicionales, a medida que el código lo requiera y simplemente cambiando el código de la clase, sin afectar al código cliente de la clase.

Herencia y derivación de clases

Vemos que el uso de clases hace más adecuada la representación de conceptos en nuestros programas. Además, **es posible generar una clase nueva a partir de otra, de la que recibe su comportamiento y estado (métodos y atributos), adaptándolos o ampliándolos según sea necesario**.

De una clase que representase el concepto de vehículo podríamos derivar otras para representar automóviles, bicicletas, barcos o aviones, donde cada uno de ellos mantiene atributos y métodos comunes (peso, color, velocidad máxima, número de pasajeros), mientras que otros son exclusivos (número de ruedas, número de hélices, número de reactores, altitud máxima de vuelo...).

En términos matemáticos podemos expresarlo diciendo que es posible establecer relaciones de pertenencia entre clases. Si tenemos una clase A y de ella derivamos una clase B, podemos decir que "*B es una A*", o que "*B es una especialización de A*".

En la terminología de la POO se dice que "*B hereda de A*", "*B es una clase derivada de A*", "*A es la clase base de B*", "*A es superclase de B*" o "*A es clase madre de B*".

Esto facilita la reutilización del código, puesto que se pueden implementar los comportamientos y datos básicos en una clase base y especializarlos en las clases derivadas.

En un programa hipotético que trabajase con formas geométricas, podríamos tener una clase base *Forma* y clases derivadas de ella como *Triangulo*, *Cuadrado*, *Circulo*... En la clase base podríamos definir un método dibuja que representa la figura en pantalla, y un método area y otro perímetro que calculan el área y el perímetro, respectivamente.

En el lenguaje python, para expresar que una clase deriva, desciende o es heredera de otra u otras clases se añade tras el nombre, en la declaración de la clase, una tupla con los nombres de las clases base.

El siguiente ejemplo crea una ClaseA y de ella deriva una ClaseB que inicializa un atributo adicional, cambia el comportamiento de un método y hereda los atributos de la clase madre:

```
class ClaseA:
    def __init__(self, x):
        self.a = x
        self.b = 2 * x
    def muestra(self):
        print "a=%i, b=%i" % (self.a, self.b)

class ClaseB(ClaseA):
    def __init__(self, x, y):
        ClaseA.__init__(self, x)
        self.c = 3 * (x + y) + self.b
    def muestra(self):
        print "a=%i, b=%i, c=%i" % (self.a, self.b, self.c)

print "Objeto A"
a = ClaseA(2)
print a.__dict__
a.muestra()

print "ObjetoB"
b = ClaseB(2, 3)
print b.__dict__
b.muestra()
```

Salida:

```
Objeto A
{'a': 2, 'b': 4}
a=2, b=4
ObjetoB
{'a': 2, 'c': 19, 'b': 4}
a=2, b=4, c=19
```

La clase ClaseA muestra funcionalidades que ya hemos ido viendo en esta introducción (atributos de instancia a y b, método muestra). La clase ClaseB se declara como descendiente de la ClaseA, por lo que hereda sus métodos y atributos y modifica algo el comportamiento de la clase madre. Por una parte, en su método de inicialización llama al método de la clase madre con los parámetros deseados, y, posteriomente, añade un nuevo atributo de instancia que no existe en la clase madre. Por otro lado, redefine un método existente en ClaseA, cambiando la implementación.

Esta última técnica, en la que un mismo nombre se asocia a comportamientos distintos se conoce como **polimorfismo**, y se habla de que el método se ha sobrecargado, es decir, se le asignan distintos significados en función de su contexto. El polimorfismo también alude a la posibilidad de utilizar, en un contexto en el que se espera una clase dada, cualquier clase derivada.

La clase object

Desde la versión *2.2* de **python**, se ha establecido una clase base llamada object de la que se aconseja derivar todas las clases. Esto permite aprovechar una serie de características sólo existentes en las "*nuevas clases*" (como la definición de propiedades), que se describen en la documentación de python.

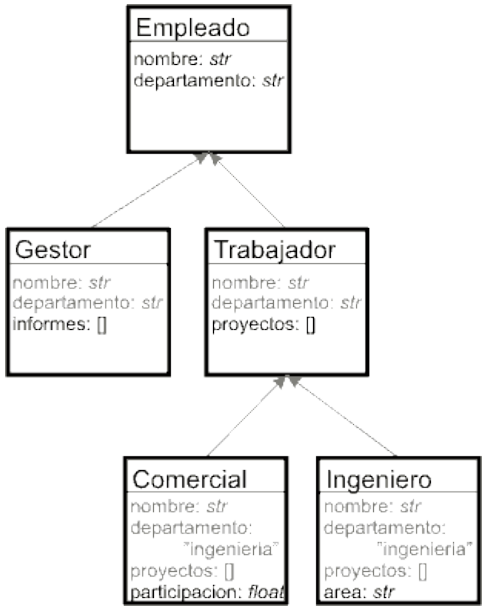
En el ejemplo sobre el uso de propiedades se introdujo la herencia sin explicarla, derivando la clase de object. Ahora debería estar más clara la razón .

Jerarquías de clases

La herencia permite establecer relaciones entre clases, y, estas relaciones de pertenencia pueden ser a varios niveles, y ramificarse en lo que se denominan **jerarquías de clases**.

Un ejemplo de clásico para visualizar las jerarquías de clases es el que podría servir para modelizar el conjunto de empleados de una empresa. En ella aparece una clase base Empleado, que dispone de los atributos nombre y departamento; de ella descienden las subclases Gestor (atributo adicional informes) y Trabajador (atributo adicional proyectos), y de esta última derivan las clases Comercial (atributo adicional participación) e Ingeniero (atributo original area).

Gráficamente podría representarse de la siguiente manera:



En el diagrama se puede ver en negrita los atributos que implementa cada clase, y en gris los que hereda de clases madre. El código (básico de inicialización) de las clases podría ser el siguiente:

```
class Empleado:
    def __init__(self, nombre, departamento="general"):
        self.nombre = nombre
        self.departamento = departamento

class Gestor(Empleado):
    def __init__(self, nombre):
        Empleado.__init__(self, nombre)
        self.informes = []

class Trabajador(Empleado):
    def __init__(self, nombre):
        Empleado.__init__(self, nombre)
        self.proyectos = []

class Comercial(Trabajador):
    def __init__(self, nombre, participacion=0.1):
        Trabajador.__init__(self, nombre)
        self.departamento = "ventas"
        self.participacion = participacion

class Ingeniero(Trabajador):
    def __init__(self, nombre, area="mecánica"):
        Trabajador.__init__(self, nombre)
        self.departamento = "ingeniería"
        self.area = area
```

Naturalmente, cada una de las clases tendría sus propios métodos que definirían comportamientos propios de cada clase o subclase, que no se han detallado en el ejemplo.

Una alternativa a la herencia es la composición, en el que una clase se compone de otras clases internas, y que puede ser una mejor alternativa en muchos casos.

Encapsulación y grados de privacidad

Uno de los principios que guían la POO, heredados de la programación modular, es el de **encapsulación**. Hemos

visto cómo se puede agrupar comportamiento y datos gracias al uso de objetos, pero hasta el momento, tanto los atributos como los métodos que se definen en las clases correspondientes se convierten en métodos y atributos visibles para los usuarios de la clase (la *API* de la clase).

Se dice que un método o atributo es **público** si es accesible desde el exterior de la clase, mientras que se denomina **privado** en caso contrario.

Es deseable poder señalar qué métodos y atributos no deben utilizarse fuera de la clase para evitar exponer excesivamente los detalles de la implementación de una clase o un objeto.

Python utiliza para ello convenciones a la hora de nombrar métodos y atributos, de forma que se señale su carácter privado, para su exclusión del espacio de nombres, o para indicar una función especial, normalmente asociada a funcionalidades estándar del lenguaje.

_nombre

Los nombres que comienzan con un único guión bajo indican de forma débil un uso interno. Además, estos nombres no se incorporan en el espacio de nombres de un módulo al importarlo con *"from ... import *"*.

__nombre

Los nombres que empiezan por dos guiones bajos indican su uso privado en la clase.

Por ejemplo, en la definición de clase siguiente:

```
class ClaseA:
    def __init__(self, a)
        self.__a = a
        self.x = self.__a**2 + 2
        self.y = self.__a**3 + 3
```

el atributo __a es de uso interno de la clase y no se exporta directamente. A continuación podemos ver cómo trata los atributos 'privados' python:

```
>>> a = ClaseA(2)
>>> dir(a)
['_ClaseA__a', '__doc__', '__init__', '__module__', 'x', 'y']
>>> a.__dict__
{'x':6, 'y':11, '_ClaseA__a':2}
```

Todavía es posible acceder a __a, pero el nombre es transformado a '_ClaseA__a', que lo señala como un atributo de uso privado.

__nombre__

Los nombres que empiezan y acaban con dos guiones bajos indican atributos "mágicos", de uso especial y que residen en espacios de nombres que puede manipular el usuario. Solamente deben usarse en la manera que se describe en la documentación de python y debe evitarse la creación de nuevos atributos de este tipo.

Algunos ejemplos de nombres "singulares" de este tipo son: __init__, método de inicialización de objetos __del__, método de destrucción de objetos __doc__, cadena de documentación de módulos, clases... __class__, nombre de la clase __str__, método que devuelve una descripción de la clase como cadena de texto __repr__, método que devuelve una representación de la clase como cadena de texto __module__, módulo al que pertenece la clase Se puede consultar la lista de atributos de este tipo y su descripción detallada en la documentación de python.

Resúmen final

La programación orientada a objetos enuncia la posibilidad de escribir un programa como un conjunto de clases de objetos capaces de almacenar su estado, y que interactúan entre sí a través del envío de mensajes.

Las técnicas más importantes utilizadas en la programación orientada a objetos son:

Abstracción: Los objetos pueden realizar tareas, interactuar con otros objetos, o modificar e informar sobre su estado sin necesidad de comunicar cómo se realizan dichas acciones.

Encapsulación (u ocultación de la información): los objetos impiden la modificación de su estado interno o la llamada a métodos internos por parte de otros objetos, y solamente se relacionan a través de una interfaz clara que define cómo se relacionan con otros objetos.

Polimorfismo: comportamientos distintos pueden estar asociados al mismo nombre

Herencia: los objetos se relacionan con otros estableciendo jerarquías, y es posible que unos objetos hereden las propiedades y métodos de otros objetos, extendiendo su comportamiento y/o especializándolo. Los objetos se agrupan así en clases que forman jerarquías.

Las clases definen el comportamiento y estado disponible que se concreta en los objetos.

Los objetos se caracterizan por:

- Tener **identidad**. Se diferencian entre sí.
- Definir su **comportamiento** a través de **métodos**.
- Definir o reflejar su **estado** a través de **propiedades y atributos**.