Programación orientada a objetos

Javier Granda

DAPI 2K 25/26

1 - ¿Qué es la programación orientada a objetos?

- Python también permite la programación orientada a objetos, que es un paradigma de la programación.

- Los datos, se agrupan por clases, atributos y/o funciones.

- Es decir, cada “clase” de dato, tendrá sus “atributos” y sus “métodos”

- *Nota aclaratoria: las clases son únicas. Objetos particulares de esa clase puede haber muchos (desde cero hasta lo que quepa en la memoria ram), y cada uno de ellos va a evolucionar de manera independiente a lo largo de la ejecución del programa.*

1.1 Las clases y sus atributos

- Una clase se define por un tipo o una tipología de dato que tiene unos atributos concretos y que todos los objetos pertenecientes a esa clase comparten.
- Por ejemplo, podemos decir, que para el tipo o “clase” de dato *Tarjeta*, se definen los siguientes “atributos”:
  - Número o PIN de tarjeta
  - Fecha de caducidad
  - Nombre o Titular de la tarjeta
  - Entidad bancaria

  Podemos decir entonces que **todos** los “objetos” pertenecientes a la “clase” de datos *Tarjeta*, tienen en común una serie de atributos que comparten y que son inherentes a esta clase de datos definida.

1.2 Las clases y sus métodos

- Los métodos de cada clase se pueden definir como las funciones que puede realizar cada clase de datos. Por ejemplo, podríamos decir que los métodos del tipo de dato *Tarjeta*, tendría los siguientes métodos o funciones:
  - *Activar tarjeta*
  - *Pagar con tarjeta*
  - *Anular tarjeta*

- Para hacer un resumen, podríamos decir que los objetos pertenecientes al tipo de dato Tarjeta tienen los siguientes atributos y funciones:

2-Acceso a los atributos y métodos de un objeto

- En Python, los tipos de datos primitivos son también objetos que tienen asociados atributos y métodos.
- Ejemplo:
- Vimos que las cadenas tienen un método upper() que convierte la cadena en mayúsculas y otro método lower() que convierte la cadena en minúsculas.
- Estos dos métodos son inherentes únicamente a esta clase de objeto
- Para aplicar este método a la cadena c se utiliza la instrucción c.upper() o c.lower().

Ejemplo. Las cadenas tienen un método *upper()* que convierte la cadena en mayúsculas. Para aplicar este método a la cadena *c* se utiliza la instrucción *c.upper()*.

```
>>> c = 'Python'
>>> print(c.upper())    # Llamada al método upper del objeto c (cadena)
PYTHON
```

Ejemplo. Las listas tienen un método append que convierte añade un elemento al final de la lista. Para aplicar este método a la lista l se utiliza la instrucción l.append(<elemento>).

```
>>> l =  [1, 2, 3]
>>> l.append(4)         # Llamada al método append del objeto l (lista)
>>> print(l)
[1, 2, 3, 4]
```

2.1 Definición de la clase de objeto

Las clases definen los miembros (constructores, atributos y métodos) y, por tanto, la semántica o comportamiento que tienen los objetos que pertenecen a esa clase. Se puede pensar en una clase como en un molde a partir del cuál se pueden crear objetos.

Para declarar una clase se utiliza la palabra clave class seguida del nombre de la clase y dos puntos, de acuerdo a la siguiente sintaxis:

```
class <nombre-clase>:
     <atributos>
     <métodos>
```

2. Definición de atributos y métodos

Los **atributos** se definen **igual que las variables** mientras que los **métodos** se definen **igual que las funciones**. Tanto unos como otros tienen que estar indentados por 4 espacios en el cuerpo de la clase.

Ejemplo. El siguiente código define la clase Saludo sin atributos ni métodos. La palabra reservada pass indica que la clase está vacía.

```
>>> class Saludo:
...     pass        # Clase vacía sin atributos ni métodos.
>>> print(Saludo)
<class '__main__.Saludo'>
```

☢ Es una buena práctica **comenzar el nombre de una clase con mayúsculas.**

3. Clases primitivas

En Python existen clases predefinidas para los tipos de datos primitivos:

- **int:** Clase de los números enteros.
- **float:** Clase de los números reales.
- **str:** Clase de las cadenas de caracteres.
- **list:** Clase de las listas.
- **tuple:** Clase de las tuplas.
- **dict:** Clase de los diccionarios.

```
>>> type(1)
<class 'int'>
>>> type(1.5)
<class 'float'>
>>> type('Python')
<class 'str'>
>>> type([1,2,3])
<class 'list'>
>>> type((1,2,3))
<class 'tuple'>
>>> type({1:'A', 2:'B'})
<class 'dict'>
```

Instanciación de clases

Para crear un objeto de una determinada clase se utiliza el nombre de la clase seguida de los parámetros necesarios para crear el objeto entre paréntesis.

- **clase(parámetros):** Crea un objeto de la clase clase inicializado con los parámetros dados.

Cuando se crea un objeto de una clase se dice que el objeto es una instancia de la clase.

```
>>> class Saludo:
...     pass        # Clase vacía sin atributos ni métodos.
>>> s = Saludo()    # Creación de objeto mediante instanciación de clase.
>>> s
<__main__.Saludo object at 0x7fcfc7756be0>   # Dirección de memoria donde se crea el objeto
>>> type(s)
<class '__main__.Saludo'>     # Clase del objeto
```

Definición de métodos

Los métodos de una clase son las funciones que definen el comportamiento de los objetos de esa clase.

Se definen como las funciones con la palabra reservada **def**. La única diferencia es que su primer parámetro es especial y se denomina **self**. Este parámetro hace siempre referencia al objeto desde donde se llama el método, de manera que para acceder a los atributos o métodos de una clase en su propia definición se puede utilizar la sintaxis **self.atributo** o **self.método.**

```
>>> class Saludo:
...     mensaje = "Bienvenido "       # Definición de un atributo
...     def saludar(self, nombre):    # Definición de un método
...         print(self.mensaje + nombre)
...         return
...
>>> s = Saludo()
>>> s.saludar('Ramón')
Bienvenido Ramón
```

La razón por la que existe el parámetro **self** es porque Python traduce la llamada a un método de un objeto **objeto.método(parámetros)** en la llamada **clase.método(objeto, parámetros)**, es decir, se llama al método definido en la clase del objeto, pasando como primer argumento el propio objeto, que se asocia al parámetro self.

El constructor __init__

En la definición de una clase suele haber un método especial llamado ***__init__*** que se conoce como constructor o inicializador. Esta función especial se llama cada vez que se instancia un objeto de la clase. Esta función inicializa los atributos que deben tener todos los objetos de la clase y por tanto contiene los parámetros necesarios para su creación, pero no devuelve nada.

```
>>> class Tarjeta:
...     def __init__(self, id, cantidad = 0):  # Inicializador
            self.id = id                        # Creación del atributo id
...         self.saldo = cantidad           # Creación del atributo saldo
...         return
...     def mostrar_saldo(self):
...         print('El saldo es', self.saldo, '€')
...         return
>>> card = Tarjeta('1111111111', 1000)      # Creación de un objeto con argumentos
>>> card.muestra_saldo()
El saldo es 1000 €
```

Atributos de Instancia vs. Atributos de Clase

Los atributos que se crean dentro del método __init__ se conocen como atributos del objeto, mientras que los que se crean fuera de él se conocen como atributos de la clase, también conocidos como campos estáticos en otros lenguajes. Mientras que los primeros son propios de cada objeto y por tanto pueden tomar valores distintos, los valores de los atributos de la clase son los mismos para cualquier objeto de la clase.

☢ Nota importante: los atributos de clase se pueden usar siempre que tengan un sentido conceptual en el diseño del programa, pero no se deben usar nunca de manera arbitraria porque sea más sencillo escribir el programa. Eso sería un error de concepto grave.

```
>>> class Circulo:
...     pi = 3.14159                    # Atributo de clase
...     def __init__(self, radio):
...         self.radio = radio          # Atributo de instancia
...     def area(self):
...         return Circulo.pi * self.radio ** 2
...
>>> c1 = Circulo(2)
>>> c2 = Circulo(3)
>>> print(c1.area())
12.56636
>>> print(c2.area())
28.27431
>>> print(c1.pi)
3.14159
>>> print(c2.pi)
3.14159
```

El método __str__

Otro método especial es el método llamado ***__str__*** que se invoca cada vez que se llama a las funciones **print** o **str**. Devuelve siempre una cadena que se suele utilizar para dar una descripción informal del objeto. Si no se define en la clase, cada vez que se llama a estas funciones con un objeto de la clase, se muestra por defecto la posición de memoria del objeto.

```
class Tarjeta:
    def __init__(self, numero, cantidad = 0):
        self.numero = numero
        self.saldo = cantidad
        return
    def __str__(self):
        return 'Tarjeta número {} con saldo {:.2f}€'.format(self.numero, str(self.saldo))

t = Tarjeta('0123456789', 1000)
print(t)
Tarjeta número 0123456789 con saldo 1000.00€
```

Conceptos importantes sobre POO

La programación orientada a objetos se refiere a algo más que tan solo atributos y métodos, también considera otros aspectos.

Dichos aspectos se conocen como **abstracción, herencia, polimorfismo, sobrecarga y encapsulamiento.**

Herencia

Una de las características más potentes de la programación orientada a objetos es la **herencia**, que **permite definir** una **especialización de una clase añadiendo nuevos atributos o métodos, o bien reescribiendo el comportamiento de algunos de ellos.** La nueva clase se conoce como subclase (o derivada, o clase hija, a veces) y hereda los atributos y métodos de la clase original que se conoce como superclase (o clase base, o padre).

Para crear un clase a partir de otra existente **se utiliza la misma sintaxis** que para definir una clase, **pero poniendo detrás del nombre de la clase entre paréntesis** el nombre de la clase madre de la que hereda.

Ejemplo. A partir de la clase Tarjeta definida antes podemos crear mediante herencia otra clase Tarjeta_Descuento para representar las tarjetas de crédito que aplican un descuento sobre las compras.

```
>>> class Tarjeta:
...     def __init__(self, id, cantidad = 0):
...         self.id = id
...         self.saldo = cantidad
...         return
...     def mostrar_saldo(self):       # Método de la clase Tarjeta que hereda la clase Tarjeta_de
...         print('El saldo es',  self.saldo, '€.')
...         return
...
>>> class Tarjeta_descuento(Tarjeta):
...     def __init__(self, id, descuento, cantidad = 0):
...         self.id = id
...         self.descuento = descuento
...         self.saldo = cantidad
...         return
...     def mostrar_descuento(self):   # Método exclusivo de la clase Tarjeta_descuento
...         print('Descuento de', self.descuento, '% en los pagos.')
...         return
...
>>> t = Tarjeta_descuento('0123456789', 2, 1000)
>>> t.mostrar_saldo()
El saldo es 1000 €.
>>> t.mostrar_descuento()
Descuento de 2 % en los pagos.
```

Una **ventaja** de la herencia, aunque no la más importante, es que **evita la repetición de código** y por tanto los **programas** son **más fáciles de mantener**.

En el ejemplo de la tarjeta de crédito, el método *mostrar_saldo()* sólo se define en la clase madre. De esta manera, cualquier cambio que se haga en el cuerpo del método en la clase madre, automáticamente se propaga a las clases hijas. Sin la herencia, este método tendría que replicarse en cada una de las clases hijas y cada vez que se hiciese un cambio en él, habría que replicarlo también en las clases hijas.

Jerarquía de clases

A partir de una clase derivada mediante herencia se pueden crear nuevas clases hijas aplicando de nuevo la herencia. Ello da lugar a una jerarquía de clases que puede representarse como un árbol donde cada clase hija se representa como una rama que sale de la clase madre.

Jerarquía de clases (CC BY-NC-SA)

Debido a la herencia, cualquier objeto creado a partir de una clase es una instancia de la clase, pero también lo es de las clases que son ancestros de esa clase en la jerarquía de clases. La “flecha” que representa la herencia en un diagrama de clases se debe leer como “es”. Es decir, en el ejemplo, una TarjetaDescuento también es una TarjetaCredito, y también es una Tarjeta. Un objeto (instancia) de una TarjetaDescuento es al mismo tiempo las tres cosas.
Al hecho de poder ver al mismo objeto de maneras diferentes, es decir, como Tarjeta, TarjetaCredito o TarjetaDescuento se le conoce como polimorfismo (= varias formas).

El siguiente comando permite averiguar si un objeto es instancia de una clase:

- **isinstance(objeto, clase):** Devuelve True si el objeto objeto es una instancia de la clase clase y False en caso contrario.

```
# Asumiendo la definición de las clases Tarjeta y Tarjeta_descuento anteriores.
>>> t1 = Tarjeta('1111111111', 0)
>>> t2 = t = Tarjeta_descuento('2222222222', 2, 1000)
>>> isinstance(t1, Tarjeta)
True
>>> isinstance(t1, Tarjeta_descuento)
False
>>> isinstance(t2, Tarjeta_descuento)
True
>>> isinstance(t2, Tarjeta)
True
```

Sobreescritura y polimorfismo

Los objetos de una clase hija heredan los atributos y métodos de la clase madre y, por tanto, a priori tienen el mismo comportamiento que los objetos de la clase madre. Pero la clase hija puede definir nuevos atributos o métodos o reescribir los métodos de la clase madre de manera que sus objetos presentan un comportamiento distinto. Esto último se conoce como **sobreescritura** (u *override*).

Si una subclase reescribe un determinado método, la versión sobreescrita de la clase padre deja de tener efecto cuando se produce la llamada. Es siempre la nueva versión de la clase hija la que se ejecuta.

En el ejemplo siguiente tenemos una Tarjeta y una TarjetaOro. La TarjetaOro es una variante especial de Tarjeta, y por tanto hereda todas sus características. Sin embargo, la TarjetaOro reescribe la forma en la que se paga con ella (método pagar).

Si instanciamos una Tarjeta “t1” (normal), al llamar a t1.pagar() se llamará a la versión definid en la clase Tarjeta.

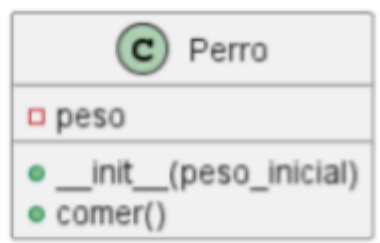
Sin embargo, si instanciamos TarjetaOro “t2”, al llamar a t2.pagar(), como TarjetaOro tiene su propia variante de pagar (overridden), será esta nueva versión la que se ejecute.

```
>>> class Tarjeta:
...     def __init__(self, id, cantidad = 0):
...         self.id = id
...         self.saldo = cantidad
...         return
...     def mostrar_saldo(self):
...         print('El saldo es {:.2f}€.'.format(self.saldo))
...         return
...     def pagar(self, cantidad):
...         self.saldo -= cantidad
...         return
>>> class Tarjeta_Oro(Tarjeta):
...     def __init__(self, id, descuento, cantidad = 0):
...         self.id = id
...         self.descuento = descuento
...         self.saldo = cantidad
...         return
...     def pagar(self, cantidad):
...         self.saldo -= cantidad * (1 - self.descuento / 1...00)
>>> t1 = Tarjeta('1111111111', 1000)
>>> t2 = Tarjeta_Oro('2222222222', 1, 1000)
>>> t1.pagar(100)
>>> t1.mostrar_saldo()
El saldo es 900.00€.
>>> t2.pagar(100)
>>> t2.mostrar_saldo()
El saldo es 901.00€.
```

Otros conceptos de la programación orientada a objetos

La programación orientada a objetos se basa en los siguientes principios:

- **Encapsulación:** proteger los atributos de un objeto, de manera que estos solo puedan ser accedidos por objetos de otra clase a través de métodos, y no de manera directa. De esta forma, el propio objeto se protege de manipulaciones erróneas, intencionadas o no, por parte de los llamadores, y solo permite las operaciones que crea oportunas. Ejemplo: perro.peso = -7 sería ilegal, y a evitar (porque no tiene sentido físico un Perro con peso negativo). Por contra, perro.comer() puede tener como efecto incrementar su peso en 100 gramos, lo que sí es coherente.

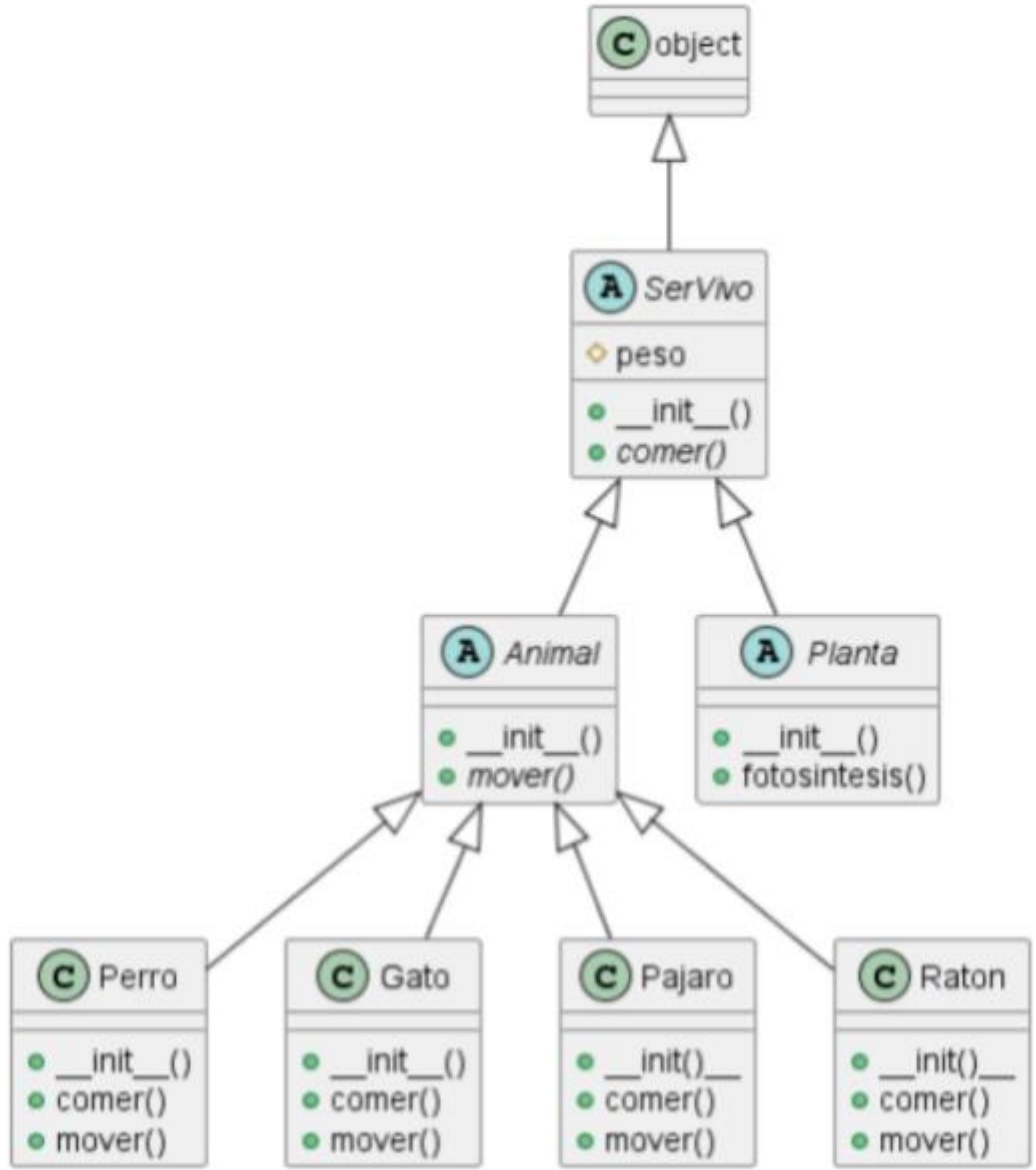


Encapsulación

Nota: en Python no existe encapsulación como tal, se deja al buen hacer de los programadores (aunque hay reglas de estilo para que estos sepan qué pueden tocar y qué no). En otros lenguajes orientados a objetos, la encapsulación es estricta: los miembros se clasifican como private, protected, package, internal, public, etc., para dar reglas rígidas de visibilidad, y el compilador garantiza que así se cumplan.

- **Herencia:** relación tipo "ser" entre clases, de manera que si la clase B "es" una variante de la clase "A", automáticamente hereda sus miembros, sin necesidad de escribirlos de nuevo en la clase B. También permite el polimorfismo, que es su uso principal.
- **Abstracción (de método):** la abstracción de método se da cuando se puede declarar un método pero no se puede definir, es decir, cuando tiene sentido pedir cierta acción a un objeto, pero no podemos codificar el cuerpo del mismo, ya que no tiene sentido o nos falta información. Por ejemplo, sabemos que cualquier Animal puede comer(), pero no podríamos implementar el cuerpo de comer(), porque no sabemos si se trata de un Perro, Pájaro, Ratón, etc. (cada uno come de forma diferente).
- **Abstracción (de clase):** una clase es abstracta si no tiene sentido instanciar objetos de dicha clase, aunque tenga ciertas características concretas.. Por ejemplo, todo Animal tiene algunas características, por ejemplo peso, volumen, sabe comer(), pero no tiene sentido instanciar un Animal(). (¿qué saldría? ¿Perro, Gato, …?).
- **Sobrecarga (overloading):** la sobrecarga se produce cuando una clase tiene varios métodos con el mismo nombre, pero estos tienen diferentes parámetros (bien en número o por ser de diferente tipo), de manera que el compilador o el intérprete pueden saber a qué método se intenta llamar, mirando los argumentos que se han inyectado en los parámetros.
- **Implementación (de método abstracto):** se denomina implementación a la primera escritura que una cierta clase hace de un método abstracto que ha heredado. Por ejemplo, todo SerVivo puede comer(), pero es algo abstracto en esta clase. El Perro ya sabe cómo come, así que el Perro implementa el método comer().
- **Sobreescritura (overriding):** la sobreescritura es la reescritura de un cierto método heredado. Perro ya tiene un método concreto para comer(), pero un Pomerania podría

- **Polimorfismo:** es la capacidad de que el llamador pueda ver a cierto objeto como de varias clases diferentes, organizadas en jerarquía. Por ejemplo, un Perro puede ser visto por otro objeto como un Perro, un Animal, un SerVivo o un Objeto.



Nota importante: Python se concibió como un lenguaje orientado a objetos, interpretado y de tipado dinámico. Esto último quiere decir que, cuando se declara una variable, Python no sabe a priori de qué clase es. De hecho, Python considera siempre a todas las instancias como pertenecientes a la clase "object". Por ejemplo, podemos crear una lista y añadirle sin mayor problema un int, un str, un Perro y una Farola, ya que la lista guarda "objects", y todas

las clases son, en última instancia, objetos. Pero este hecho tiene sus puntos débiles: Python no es un lenguaje puro orientado a objetos, porque el tipado dinámico hace que la abstracción (que no tiene) y el polimorfismo no se puedan utilizar como dicta la teoría de este paradigma. Tampoco tiene, por los mismos motivos, sobrecarga ni implementación (sí sobreescritura).

Conclusiones

- Resolver un problema siguiendo el paradigma de la programación orientada a objetos requiere un cambio de mentalidad con respecto a cómo se resuelve utilizando el paradigma de la programación estructurada. Sin embargo, no hay que olvidar que la programación estructurada sigue siendo el corazón de la programación orientada a objetos.
- La programación orientada a objetos es más un proceso de modelado, donde se identifican las entidades que intervienen en el problema y su comportamiento, y se definen clases que modelizan esas entidades. Por ejemplo, las entidades que intervienen en el pago con una tarjeta de crédito serían la tarjeta, el terminal de venta, la cuenta corriente vinculada a la tarjeta, el banco, etc. Cada una de ellas daría lugar a una clase.
- Se crean objetos con los datos concretos del problema y se hace que los objetos interactúen entre sí, a través de sus métodos, para resolver el problema. Cada objeto es responsable de una subtarea y colaboran entre ellos para resolver la tarea principal. Por ejemplo, la terminal de venta accede a los datos de la tarjeta y da la orden al banco para que haga un cargo en la cuenta vinculada a la tarjeta.
- De esta forma se pueden abordar problemas muy complejos descomponiéndolos en pequeñas tareas que son más fáciles de resolver que el problema principal (¡divide y vencerás!).