Fundamentos de Programación

Tema 8. Introducción a la orientación a objetos

José Francisco Díez Pastor

24 de marzo de 2020

Área de Lenguajes y Sistemas Informáticos
Departamento de Ingeniería Civil - Universidad de Burgos

Índice de contenidos

1. ¿Qué vamos a ver?

¿Qué vamos a ver?


En esta sesión vamos a ver una introducción a la programación orientada a objetos. Veremos qué son las clases, qué son las instancias, los métodos etc

Y aprenderemos algunas características básicas de la orientación a objetos como la herencia, la encapsulación o el polimorfismo.

2. ¿Qué es la programación orientada a objetos?

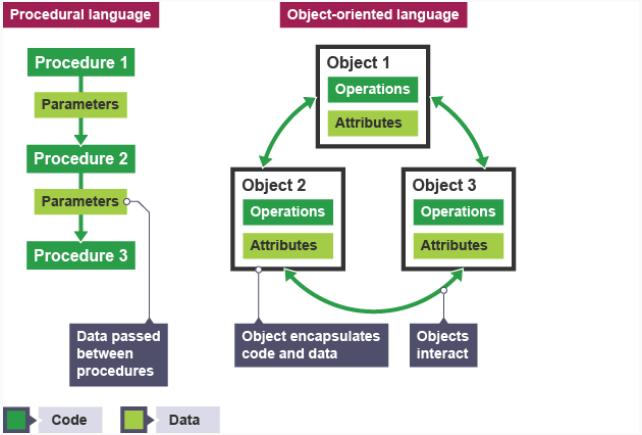
¿Qué es la programación orientada a objetos?


La programación orientada a objetos es un paradigma de programación (alternativo a la programación estructurada o la programación funcional) en el que se llega a la solución mediante la descomposición del problema en objetos. Posteriormente estos objetos interactúan entre si del modo que define el programador.

Por ejemplo, un objeto puede representar a una persona que tiene como atributos: edad, dirección, etc y que tiene comportamientos como: hablar, andar, respirar etc.

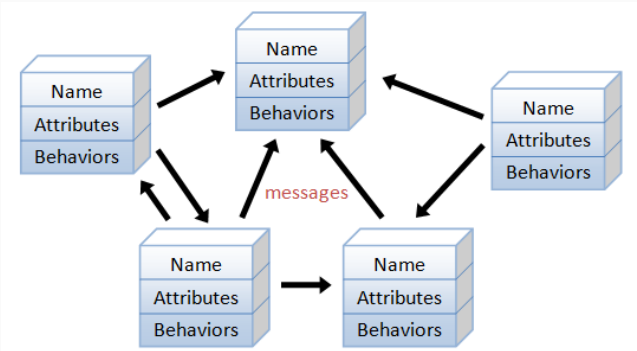
Otro ejemplo, un objeto *email* puede tener atributos como: lista de destinatarios, asunto, cuerpo etc y como comportamientos puede tener: añadir adjunto o enviar.

¿Qué es la programación orientada a objetos?



Los objetos encapsulan datos y métodos e interactuan unos con otros.

¿Qué es la programación orientada a objetos?



An object-oriented program consists of many well-encapsulated objects and interacting with each other by sending messages

Los objetos encapsulan datos y métodos e interactuan unos con otros.

¿Qué es la programación orientada a objetos?


La programación orientada a objetos es apropiada cuando queremos modelar cosas concretas del mundo real que están relacionadas entre si, cosas como coches, compañías, empleados, estudiantes, profesores, pacientes, medicamentos etc.

En programación orientada a objetos, los objetos son entidades que tienen asociadas unos datos (atributos) y que pueden realizar unas operaciones (métodos).

Nota

Como Python es un lenguaje multiparadigma, se puede elegir el paradigma de programación que mejor nos venga en cada caso.

En un mismo programa se pueden emplear elementos de distintos paradigmas de programación, por ejemplo podemos emplear programación orientada a objetos y programación funcional.

Clase Las clases se utilizan para crear tipos de datos definidos por el usuario. Permiten mantener en un mismo sitio un conjunto de atributos junto con los métodos que trabajan con esos datos.

Objeto Los objetos son instancias o copias particulares de una clase, con unos valores determinados para los atributos.

La clase es el «molde» y el objeto es el caso particupar que se crea con dicho molde.

Ejemplo:

- Podemos tener una clase Persona.
- Con la clase Persona creamos los objetos: Pepe, Juan y Maria.

Para definir una clase se utiliza la palabra reservada `class` seguida por el nombre de la clase y dos puntos.

A continuación hay una serie de métodos, todos ellos tienen parámetro especial (`self`). Luego al invocar al método no se emplea.

Un método especial es el constructor, que crea el objeto e inicializa los atributos. El constructor es un método con nombre `__init__`.

```
class nombre_clase:
    def __init__(self,argumentos):
        # inicializacion
        ...

    def metodo1(self,argumentos):
        # implementacion del metodo 1
    ...
```

Para crear un objeto invocamos el nombre de la clase seguido de paréntesis.

```
objeto1 = nombre_clase()
```

Clases y objetos en Python

```python
class Gato:

    # Constructor, inicializa los datos
    def __init__(self, nombre, edad):
        self.nombre = nombre
        self.edad = edad

        self.energia = 10

    def maulla(self):
        print(f"Miau, soy {self.nombre}")

    def duerme(self):
        self.energia = 10

    def juega(self):
        if self.energia>0:
            print("Miauu miauu")
            self.energia -= 5
        else:
            print("Grrrr")
```

```python
gato1 = Gato("Calcetines",3)

gato1.maulla()
gato1.juega()
gato1.juega()
gato1.juega()
gato1.juega()

gato1.duerme()

gato1.juega()
```

```
Miau, soy Calcetines
Miauu miauu
Miauu miauu
Grrrr
Grrrr
Miauu miauu
```

Defino una clase **Gato**.

- El constructor tiene 3 parámetros: self, nombre y edad.
- El resto de métodos solo tiene 1 parámetro: self.
- Al invocar esos métodos no se usa self.
- El constructor crea un gato con un nombre, una edad y 10 de energía.
- El método maulla le hace decir su nombre.
- El método duerme le repone la energía.
- Con el método juega, maulla y gasta energía, si no tiene energía gruñe.

El método `__init__` es un método especial que tienen las clases en Python.

El método `__init__`:

- Sirve para inicializar los atributos de los objetos que creamos.
- Es el método que se invoca automáticamente en el momento de crear el objeto.
- No retorna nada.
- Puede recibir parámetros, en ese caso tenemos que usar dichos parámetros al crear el objeto.
- Es opcional, podemos no declarar el método `__init__`, pero lo normal es hacerlo. Si no tenemos un método `__init__`, al crear el objeto no inicializamos ninguna variable.

Variables de clase vs Variables de instancia

Una variable de clase **es común para todos los objetos de la misma clase**, cuando se actualiza su valor se actualiza para todos los objetos de esa clase.

Una variable de instancia **puede tener un valor diferente para cada objeto creado**. Los valores se almacenan en el propio objeto.

Las variables de instancia de dos objetos diferentes son variables independientes, aunque tengan el mismo nombre, porque se almacenan en objetos diferentes.

Una variable de clase se define dentro del bloque `class` pero fuera de cualquier método.

Una variable de instancia se crea dentro de un método, habitualmente el constructor.

Método de clase vs método de instancia

Al igual que hay variables de clase y variables de instancia ocurre lo mismo con los métodos.

Los métodos de clase se utilizan para acceder o modificar valores que son comunes a todos los objetos de una clase.

Los métodos de instancia utilizan valores que son propios de cada una de las instancias (variables de instancia).

La palabra clave `self` representa una instancia concreta de la clase.

Mediante `self` se puede acceder a atributos y a métodos de instancia. Es decir a los atributos y métodos de un objeto concreto.

Cuando se define un método de instancia el primer parámetro siempre tiene que ser la palabra clave `self`.

Método de clase vs método de instancia

- Métodos y atributos de instancia.
  - Utilizo la palabra reservada `self`.
- Métodos y atributos de clase.
  - Hago referencia al propio nombre de la clase.

Ver el ejemplo siguiente.

Clases y objetos en Python.

```python
class Persona:
    apellido = "Fernandez"

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

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

    def cambia_apellido(apellido):
        Persona.apellido=apellido

    def decir_nombre(self):
        print(f"Soy {self.nombre} {Persona.apellido}")
```

```python
p1 = Persona("Pepe")
p2 = Persona("Juan")

p1.cambia_nombre("Manuel")

p1.decir_nombre()
p2.decir_nombre()

Persona.cambia_apellido("Rodiguez")

p1.decir_nombre()
p2.decir_nombre()
```

```
Soy Manuel Fernandez
Soy Juan Fernandez
Soy Manuel Rodiguez
Soy Juan Rodiguez
```

__str__ Método que al invocarlo devuelve una representación en modo texto de la clase. Se invoca al hacer algo como esto `print(objeto)`

__repr__ Método que al invocarlo devuelve una representación abreviada en modo texto de la clase. Se invoca al imprimir una colección que contenga objetos de ese tipo `print([objeto1, objeto2])`

__eq__ Método que recibe otro objeto y devuelve True si el objeto pasado como argumento es igual al objeto actual o False en caso contrario.

__hash__ Método que debemos implementar si queremos usar este tipo de objeto como clave en un diccionario o como elemento de un conjunto. Es un método que debe devolver un entero. Una forma fácil de crear un hash es obteniendo el hash de un tupla formada por todas las variables de instancia del objeto que se usen en `__eq__`.



```python
class Coche:
    """
    Los comentarios con triple comilla son comentarios de clase o metodos
    pueden ocupar varias lineas.
    """
    def __init__(self,nombre):
        self._nombre = nombre
        self._velocidad = 0

    def acelera(self):
        self._velocidad=self._velocidad+1

    def frena(self):
        self._velocidad=0

    def __str__(self):
        return f"El {self._nombre} va a {self._velocidad} km/h"

    def __repr__(self):
        return f"{self._nombre} ({self._velocidad}km/h)"

    def __eq__(self,other):
        return self._nombre == other._nombre

    def __hash__(self):
        return hash((self._nombre))
```

Clases y objetos en Python. Métodos especiales

```
coche1=Coche("Renault")
coche2=Coche("Seat")

print(coche1)
print(coche2)

coche1.acelera()
coche1.acelera()
coche1.acelera()
coche2.acelera()

print(coche1)
print(coche2)
coche1.frena()

print(coche1)
print(coche2)
```

```
Renault va a 0 km/h
Seat va a 0 km/h
Renault va a 3 km/h
Seat va a 1 km/h
Renault va a 0 km/h
Seat va a 1 km/h
```

```
lista = [coche1,coche2]
print(lista)
```

```
[Renault (0km/h), Seat (1km/h)]
```

El método __eq__ que se ha definido en el ejemplo solo utiliza como campo de comparación el nombre del coche, así que dos coches son iguales si tienen la misma marca, independientemente de la velocidad a la que vayan.

```
coche1==coche2
```

```
False
```

```
coche3=Coche("Renault")
coche1==coche3
```

```
True
```

El método `__hash__` se utiliza para añadir elementos a un conjunto. Como el coche1 y el coche3 son iguales de acuerdo a nuestro ejemplo y tienen el mismo hash, al añadirlos a un conjunto solo se almacena el primero de ellos.

```
# Se usa el método hash y el método repr
set([coche1,coche2,coche3])
```

```
{Renault (0km/h), Seat (1km/h)}
```

Si queremos añadir un objeto a un conjunto o como clave de un diccionario y no hemos implementado el método `__hash__` obtenemos un error como el siguiente:

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-38-b416246e1313> in <module>
      1 # Se usa el método hash y el método repr
----> 2 set([coche1,coche2,coche3])

TypeError: unhashable type: 'Coche'
```

3. Características principales de la POO (Programación orientada a objetos)

Existen una serie de características que comparten prácticamente todos los lenguajes orientados a objetos:

- Encapsulación.
- Abstracción.
- Herencia.
- Polimorfismo.

Abstracción

La abstracción es el proceso de seleccionar características relevantes y comportamientos comunes a todos los elementos de un mismo tipo.

La abstracción es clave en el análisis y diseño orientado a objetos, ya que permie modelar el mundo real o el problema a resolver usando un conjunto de clases.

Los objetos tienen una serie de propiedades y comportamientos y están relacionados con otros objetos (pueden llamar a métodos de otros objetos). Un objeto puede comunicarse con otro, consultar datos, cambiar su estado etc sin preocuparse de como están implementados los métodos que usa.

Las clases no están aisladas, sino que se relacionan entre si. No solo una clase puede usar objetos de otra clase, sino que las clases se pueden organizar formando una jerarquía.

En el nivel más alto de la jerarquí tenemos una *superclase* de la que otras clases, llamadas *subclases* heredan ciertos atributos y métodos.

Una subclase puede heredar propiedades y comportamientos de todas las clases de las que hereda, no solo de aquella que está inmediatamente a un nivel superior.

Cuando una clase hereda de varias clases a la vez se denomina herencia múltiple.

Para heredar en Python hay que indicar entre paréntesis el nombre de la clase de la que se quiere heredar, en el momento de definir la clase.

Con esto se heredan todas las variabres y métodos, si queremos redefinir uno podemos hacerlo simplemente volviendolo a definir.

```
class CocheRapido(Coche):
    """
    Coche rápido que acelera mucho más rápido
    """
    def acelera(self):
        self._velocidad=self._velocidad+3

coche1=Coche("Renault")
coche2=CocheRapido("Ferrari")
```

```
coche1=Coche("Renault")
coche2=CocheRapido("Ferrari")

print(coche1)
print(coche2)

coche1.acelera()
coche1.acelera()
coche2.acelera()
coche2.acelera()

print(coche1)
print(coche2)
```

```
El Renault va a 0 km/h
El Ferrari va a 0 km/h
El Renault va a 2 km/h
El Ferrari va a 6 km/h
```

Hay dos funciones que trabajan con herencia en python:

- `isinstance()` Devuelve True si el primer argumento es de la clase indicada por el segundo argumento.
- `issubclass()` Devuelve True si la clase del primer argumento es subclase de la clase del segundo argumento.

```
print(isinstance(coche1, Coche))
print(isinstance(coche1, CocheRapido))
```

```
True
False
```

```
print(issubclass(Coche, CocheRapido))
print(issubclass(CocheRapido,Coche))
```

```
False
True
```

Herencia múltiple en Python

Para hacer herencia múltiple en Python simplemente indicamos las clases de las que se quiere heredar y automáticamente podemos usar los métodos y propiedades definidos en cualquiera de las clases padre.

```python
class Scanner:
    def scanea(self):
        print("Escaneo realizado")

class Impresora:
    def imprime(self):
        print("Impresión realizada")

class Copiadora(Scanner,Impresora):
    def copia(self):
        self.scanea()
        self.imprime()

copiadora = Copiadora()
copiadora.copia()
```

```
Escaneo realizado
Impresión realizada
```

Clases abstractas

Una clase abstracta contiene uno o varios métodos abstractos, métodos que no están implementados.

Permite definir una serie de métodos que deben ser implementados por todas sus subclases (o clases hijas).

Una clase abstracta puede considerarse como un plano o esquema para otras clases. No pueden instanciarse directamente, sino que es necesario instanciar una de las subclases.

Clases abstractas. ¿Para qué sirven?

Las clases abstractas permiten definir una funcionalidad por defecto que tienen que tener todas las clases. Si una clase hija de una clase abstracta no implementa todos los métodos abstractos no podría instanciarse.

Con una clase abstracta puede definirse un **API** (*Application Program Interface*) que deben cumplir las subclases. Esto es especialmente útil para permitir que terceros implementen código compatible con una aplicación (por ejemplo plugins) o para trabajar en grandes equipos.

Clases abstractas en Python

Python no proporciona por defecto clases abstractas, pero tiene un módulo que proporciona dicha funcionalidad, se trata del módulo **ABC** (*Abstract Base classes*).

- Se puede indicar que una clase es abstracta heredando de ABC.
- Se puede indicar que un método es abstracto «decorandolo» con la palabra clave `@abstractmethod`.

Clases abstractas en Python

```python
from abc import ABC, abstractmethod

class Plugin(ABC):

    @abstractmethod
    def configure(self):
        pass

    @abstractmethod
    def execute(self):
        pass

class PluginDropBox(Plugin):

    def configure(self,login,password):
        print("Conectando")

    def execute(self):
        print("Sincronizando")

dropbox = PluginDropBox()
dropbox.configure("user","pass")
dropbox.execute()
```

```
Conectando
Sincronizando
```

Clases abstractas en Python

Si una subclase de una clase abstracta no implementa todos los métodos abstractos no se puede instanciar (no se pueden crear objetos de dicha clase).

```
class PluginDropBoxFail(Plugin):

    def configure(self,login,password):
        print("Conectando")


dropbox_fail = PluginDropBoxFail()
```

```
---------------------------------------------------------------------------
-----
TypeError                                 Traceback (most recent call
last)
<ipython-input-42-b8e50fcda0d2> in <module>
      5
      6
----> 7 dropbox_fail = PluginDropBoxFail()

TypeError: Can't instantiate abstract class PluginDropBoxFail with abs
tract methods execute
```

Encapsulación

La **encapsulación** es una de las características fundamentales de la programación orientada a objetos, se refiere a la capacidad de agrupar los datos junto con los metodos que trabajan con dichos datos.

La encapsulación permite ocultar variables de una clase, para que otras clases no puedan acceder a esos valores.

Se definen 3 tipos de niveles:

Publico Una variable o método público puede ser usado desde cualquier parte del programa.

Protegido Una variable o método protegido solo puede ser usada desde la propia clase o clases que hereden de ella.

Privado Una variable o método protegido solo puede ser usada desde la propia clase.

```java
public class Dog extends Animal {

    private int numberOfLegs;
    private boolean hasOwner;

    public Dog() {
        numberOfLegs = 4;
        hasOwner = false;
    }

    public void makeDogBark() {
        Dog d = new Dog();
        d.bark();
    }

    private void bark() {
        System.out.println("Woof!");
    }

    public void move() {
        System.out.println("Running");
    }

}
```

Encapsulación en Java.

Encapsulación en Python

```python
class Robot:
    def __init__(self):
        self.marca = "Sony"
        self._codigo_protegido = "123"
        self.__codigo_privado = "Secreto"

    def __proceso_privado(self):
        print("Hace cosas")

    def _proceso_protegido(self):
        self.__proceso_privado()

class Robot_Movil(Robot):

    def funciona(self):
        self._proceso_protegido()

rm1 = Robot_Movil()
rm1.funciona()
```

```
Hace cosas
```

Encapsulación en Python

```
class Robot_Movil(Robot):

    def funciona(self):
        self.__proceso_privado()

rm1 = Robot_Movil()
rm1.funciona()
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-44-4d9d24ca6ec7> in <module>
     17
     18 rm1 = Robot_Movil()
---> 19 rm1.funciona()

<ipython-input-44-4d9d24ca6ec7> in funciona(self)
     14
     15     def funciona(self):
---> 16         self.__proceso_privado()
     17
     18 rm1 = Robot_Movil()

AttributeError: 'Robot_Movil' object has no attribute '_Robot_Movil__proceso_privado'
```

Encapsulación en Python

```
class Controlador_Robots:

    def controla(self):
        r1 = Robot()
        print(r1.marca)
        # protected no funciona en python
        # se puede acceder igual
        print(r1._codigo_protegido)
        #print(r1.__codigo_privado)


c1 = Controlador_Robots()
c1.controla()
```

```
Sony
123
```

Polimorfismo, en el contexto de la programación, se refiere a la capacidad de que un mismo método esté asociado a comportamientos diferentes.

O dicho de otro modo, podemos tener varios objetos y al invocar a un método por su nombre, siempre tendrá el comportamiento correcto que se corresponde a su clase.

```python
from abc import ABC, abstractmethod

class Forma(ABC):

    @abstractmethod
    def calcula_area(self):
        pass

class Triangulo(Forma):

    def __init__(self,base,altura):
        self.base = base
        self.altura = altura

    def calcula_area(self):
        return self.base * self.altura / 2
```

Se está usando una clase abstracta, pero no sería obligatorio.

```python
class Rectangulo(Forma):

    def __init__(self,base,altura):
        self.base = base
        self.altura = altura

    def calcula_area(self):
        return self.base * self.altura
```

```
#f = Forma()
t1 = Triangulo(10,5)
t2 = Triangulo(20,15)
t3 = Triangulo(10,8)
r1 = Rectangulo(20,10)
r2 = Rectangulo(10,18)

formas = [t1,t2,t3,r1,r2]
for forma in formas:
    print(forma.calcula_area())
```

```
25.0
150.0
40.0
200
180
```