Programación Orientada a Objetos

Sitio: Espacio de Formación EPCIA
Curso: Programación con Python. Programación Orientada a Objetos
Libro: Programación Orientada a Objetos
Imprimido por: Salim Tieb Mohamedi
Día: jueves, 18 de noviembre de 2021, 01:30

Tabla de contenidos

1. Introducción

En este punto tenemos ya una base de Python donde conoces los tipos de datos básicos que puedes manejar en Python:

- boolean, que representan un valor verdadero (True) o falso (False),
- integer, para representar valores numéricos enteros (1,4,7,…)
- float, para valores numéricos no enteros (1.465, 4.232, …),
- complex, para valores numéricos complejos (1 + 5j, 6 -3j, ...), y
- string, para cadenas de texto (“hola”, “adios”).

Asignando valores de estos tipos básicos a variables y manipulándolos mediante operadores y expresiones del lenguaje, puedes resolver cualquier problema que se te plantee. Sin embargo, cuando el problema alcanza cierta complejidad, resolverlo usando solamente estos tipos de datos básicos puede ser bastante difícil. Por eso, los lenguajes de programación, Python entre ellos, ofrecen también algunos tipos de datos que se obtienen mediante combinación o composición de los anteriores. Nos referimos a: las listas, las tuplas y los diccionarios. Todos ellos se obtienen agrupando de alguna forma valores del mismo o distintos tipos.

Un ejemplo de lista: [5, 7, “hola”, True].

Un ejemplo de tupla: (5, 7, 3).

Un ejemplo de diccionario: {“nombre”: “Pepe”, “apellido”: “Grillo”}

Estos tipos, que suelen denominarse tipos compuestos, permiten organizar los datos de manera que resulte más sencillo resolver problemas en los que se necesita manejar una gran cantidad de información.

Para saber más

Si quieres conocer en detalle la diferencia entre tipos de datos como listas, tuplas y diccionarios consulta este artículo:

https://python-para-impacientes.blogspot.com/2014/01/cadenas-listas-tuplas-diccionarios-y.html

Pero una vez más, cuando el problema que tenemos entre mano implica la representación de conceptos más o menos complejos, que además se relacionan entre sí de acuerdo a ciertas reglas, esos tipos de datos se nos pueden quedar cortos para organizar de una manera eficaz toda la información que debemos manejar. Y es aquí donde entra la *Programación Orientada a Objetos (POO)*, una manera eficaz de concebir, organizar y expresar los datos y operaciones en los programas para resolver problemas no triviales.

Los lenguajes de programación modernos como Python, Java, Javascript, PHP, C++, Scala, Lua, ofrecen la posibilidad de organizar los datos mediante los principios de la POO. Eso sí, cada uno lo hace a su manera y siguiendo distintos mecanismos; algunos como Python, Java y C++ se basan en el concepto de **clase**, y otros, como Javascript o Lua en el de **prototipo**; dos maneras distintas de entender la representación de datos que están encaminadas a resolver el mismo problema: organizar el código más eficazmente facilitando su comprensión, mantenimiento, extensión y reutilización.

1.1. La función type()

La función **type()** de Python, admite como único argumento un dato de cualquier tipo, y devuelve como resultado el tipo al que pertenece dicho dato. Compruébalo tú mismo usando el interprete interactivo IDLE de Python:

```
>>> type("hola caracola")
<class 'str'>
>>> type(4+8j)
<class 'complex'>
>>> type([5,7,8])
<class 'list'>
```

Consejo: aplica la función type() a valores de todos los tipos que conoces, así verás el nombre que Python asigna internamente a cada tipo de dato.

Fíjate que en todos los resultados anteriores, el nombre del tipo de dato está precedido por la palabra *class* (clase en inglés). La forma correcta de leer las líneas anteriores sería:

- "hola caracola" **es** un **objeto** de la **clase** *str*,
- 4 + 8j **es** un **objeto** de la **clase** *complex*,
- [5,7,8] **es** un **objeto** de la **clase** *list.*

Las frases anteriores tiene un sentido fácilmente comprensible sin necesidad de saber mucho sobre programación de computadoras, pues son expresiones que usamos habitualmente en nuestro lenguaje (por ejemplo, mi citroën berlingo **es** un **objeto** de la **clase** coche). Lo bueno es que bajo la expresión "*x **es** un **objeto** de la **clase** X*", subyace toda la base de la Programación Orientada a Objetos (en adelante POO) basada en clases. Y eso es lo que trabajaremos a lo largo de este módulo.

Acabamos de ver que cualquier tipo de datos en Python es un objeto, y que todos los objetos pertenecen a una clase (al menos a una, ya veremos más adelante que un mismo objeto puede pertenecer a varias). Pues bien, la POO nos permite crear nuestras propias clases de objetos, o dicho de otro modo, nuestros propios tipos. Y por eso es particularmente útil cuando queremos representar conceptos de la vida real, es decir, mediante la POO conseguimos crear tipos de datos que **modelan** la estructura y el comportamiento de conceptos reales, tales como el de *persona*, *dirección postal*, *figura geométrica*, *factura*, *pedido*, *libro*, *coche*, *motor*, etcétera. Vamos, cualquier cosa que de una u otra forma aparezca en el problema que nos proponemos resolver mediante un programa informático.

En uno de los libros más influyentes sobre programación de computadoras, "*Structure and interpretation of computer programs*"[1], encontramos una de las ideas más potentes sobre los lenguajes de programación:

"*A powerful programming language is more than just a means for instructing a computer to perform tasks. The language also serves as a framework within which we organize our ideas about processes.*"

Es decir, que la razón de ser de un lenguaje de programación va más allá de servir como instrumento para programar a la máquina, es también un medio para que expresemos nuestras ideas, y por tanto debe proporcionar mecanismos de comunicación cercanos al lenguaje humano. La POO es, sin lugar a dudas, uno de los intentos más conseguidos de dotar a los lenguajes de programación de características cercanas al lenguaje humano, con las que poder modelar los problemas del mundo externo de una manera cercana a nuestro modo de pensar.

1Abelson, H., & Sussman, G. J. (1996). *Structure and interpretation of computer programs* (p. 688). The MIT Press.

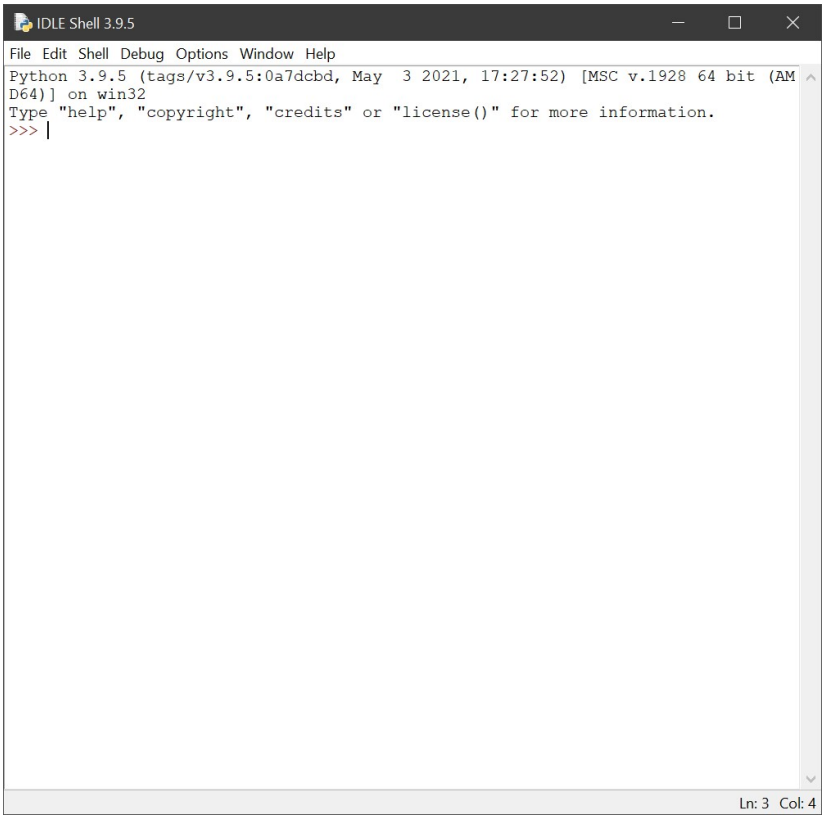
1.2. 1.2. Trabajar con Python

Si estamos trabajando este contenido es porque ya tenemos una base adecuada de Python y seguramente tengamos instalado nuestro propio Entorno Integrado de Desarrollo (IDE) de Python.

No obstante, si no es así. Antes de comenzar a trabajar necesitaremos un interprete de Python instalado. En Linux esto no es necesario porque ya lo tenemos instalado. Si vas a trabajar con Windows necesitarás un interprete. Lo más sencillo es descargarlo de su página web:

https://www.python.org/downloads/

Una vez instalado, en el menú Inicio encontrarás, junto al interprete un pequeño entorno de desarrollo que nos servirá para realizar los ejemplos que veremos a continuación, **IDLE**.



Podemos escribir comandos directamente en IDLE Shell o bien mediante el menú **File** podremos crear nuevos archivos donde almacenar nuestros programas y probarlos.

Si lo deseas puedes explorar otros entornos aún más completos, como JetBrain PyCharm.


JET BRAINS
Para desarrolladores
Para equipos
Para aprender
Soluciones
Tienda
PyCharm
Próximamente en 2021.3
Novedades
Funcionalidades
Aprender
Comprar
Descargar
PC
PyCharm
IDE de Python
para desarrolladores
profesionales
DESCARGAR
Versión Professional completa o versión gratis Community

2. Clases y objetos

Piensa por un momento en coches concretos, en los que ves cuando sales a la calle. Todos representan un tipo de objeto al que llamamos, de forma genérica, *coche*. El *coche*, en este sentido, no es algo material, es una idea en la que agrupamos a objetos que comparten unas mismas características y un comportamiento similar; son objetos que tienen 4 ruedas, puertas, un motor, un chasis, se desplazan a una velocidad que se puede controlar, etcétera. Cada uno de los coches concretos que ves en la calle son, por tanto, **objetos** de una misma **clase** que llamamos *coche*. Y esto ocurre con gran parte de las cosas que conocemos, desde las más físicas y tangibles, como el *coche*, hasta las más etéreas y abstractas como una figura geométrica. Es algo que tiene que ver con la forma en que construimos el conocimiento general a partir de los casos concretos.

En Python (y en todos los lenguajes que permiten usar la POO basada en clases), la clase es una plantilla que define los atributos (datos) y el comportamiento (funciones) que tendrán los objetos creados a partir de ella. Usando una analogía, podemos comparar el concepto de clase con un molde a partir del cual se obtienen soldaditos de plomo. Aunque todos ellos son iguales entre sí, cada uno es una pieza distinta. Otra analogía puede ser la de un formulario sin rellenar que fotocopiamos cada vez que necesitamos recoger los datos de un nuevo socio de la biblioteca. El formulario patrón sin rellenar sería la clase, y las fotocopias, que rellenamos cada una con unos datos distintos, serían los objetos.

Como el camino se demuestra andando, vamos a crear nuestra primera clase en Python. Primero pensemos en algún concepto que modelar. Supongamos que vamos a escribir un programa para gestionar el funcionamiento de una biblioteca. Uno de los conceptos que aparecerán con toda seguridad será el de *Libro*. Así que comenzaremos creando una clase para modelar la idea genérica de *Libro*. Es decir, para extraer las características más importantes que tiene cualquier *libro* en el ámbito de una biblioteca.

Abre el editor de IDLE, y en un fichero al que puedes llamar libro.py escribe lo siguiente:

```
class Libro():
    pass
```

Esta es la clase más sencilla que podemos escribir. Y aunque aún no recoja ninguna característica de lo que es un libro, ya es funcional. Es decir, ya se puede usar en un programa Python. Como ves, para definir una clase se utiliza la palabra clave class seguido del nombre de la clase, unos paréntesis y el carácter ":". Ya sabes que en Python, los dos puntos sirven para indicar el comienzo de un bloque. En este caso se usan para indicar el bloque que alojará el código de la clase. Por lo pronto solo contiene la palabra clave pass, la cual, tal y como sugiere su significado, pasa de todo y no hace nada, pero rellena el bloque y evita que se produzca un error al ejecutar el código. En breve sustituiremos esta palabra clave por el código de la clase, pero antes vamos a ver como se crean objetos a partir de una clase.

Escribe el siguiente código y ejecútalo.

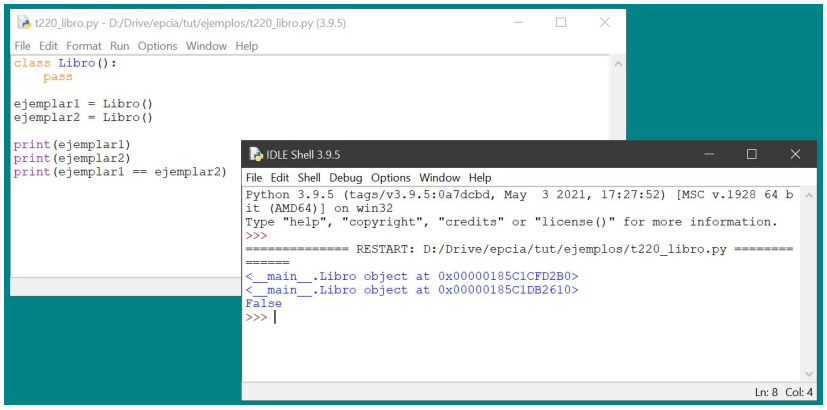
```
class Libro():
    pass

ejemplar1 = Libro()
ejemplar2 = Libro()

print(ejemplar1)
print(ejemplar2)
print(ejemplar1 == ejemplar2)
```

Obtendrás como resultado lo siguiente:

```
<__main__.Libro object at 0x110073ac0>
<__main__.Libro object at 0x1101040d0>
False
```



Aunque en tu caso los número hexadecimales serán distintos a los anteriores. Lo que está diciendo el resultado de print(ejemplar1), es decir, <__main__.Libro object at 0x110073ac0>, es que la variable ejemplar1 es un objeto de la clase Libro que se ha almacenado en la dirección de memoria 0x110073ac0. Por su parte el resultado de print(ejemplar2), es decir, <__main__.Libro object at 0x1101040d0>, indica que ejemplar2 es también un objeto de la clase Libro, pero alojado en otra dirección de memoria distinta. Esto significa que ambas variables están apuntando a distintos objetos. La última línea del resultado corrobora este hecho, pues la salida de la expresión booleana ejemplar1 == ejemplar2 es False, expresando claramente que ambas variables apuntan a distintos objetos.

Observa que ambas variables se han creado asignándolas al resultado de invocar a la clase Libro.

```
ejemplar1 = Libro()
ejemplar2 = Libro()
```

Aunque parece que estamos asignando las dos variables a la misma cosa, como acabamos de ver, esto no es así. La conclusión es que cada vez que invocamos a la clase Libro seguida de unos paréntesis, se crea un nuevo objeto en algún lugar del espacio de memoria. Por eso al nombre de la clase seguida de los paréntesis se le conoce como **constructor** de la clase. En el intérprete interactivo de IDLE, prueba a invocar la clase Libro muchas veces. Verás como cada vez obtienes un objeto distinto en una nueva posición de memoria.

```
>>> Libro()
<__main__.Libro object at 0x1100ef880>
>>> Libro()
<__main__.Libro object at 0x1100ef850>
>>> Libro()
<__main__.Libro object at 0x1100ef280>
```

Así que ya sabemos como crear nuevos objetos a partir de una clase. Basta con invocarla como si se tratase de una función. Al proceso de creación de objetos a partir de una clase se le llama frecuentemente *instanciar* una clase. Y a los objetos creados se les llama también *instancias* de una clase.

Ejercicio 1: ¿Qué resultado obtendremos si aplicamos la función type() a las variables ejemplar1 y ejemplar2?

Solución: Como la función type() devuelve el tipo al que pertenece el valor de su argumento, y tanto ejemplar1 como ejemplar2 son objetos de la clase Libro, el resultado en ambos casos será <class '__main__.Libro'>

2.1. La función __init__()

Por lo pronto nuestra clase es bastante insulsa, pero nos ha servido para saber como se declara una clase y como se crean objetos a partir de ella. Ahora vamos a añadir algo de código para capturar lo más relevante del concepto Libro.

Nota: “lo más relevante” dependerá del problema concreto que estemos resolviendo. Por ejemplo, el precio el libro es una característica relevante para la gestión de una tienda de libros, pero no lo es tanto para la de una biblioteca. Esto es la esencia del modelado, extraer aquellas cosas que realmente influyen en el problema que tratamos de resolver.

Cuando se invoca una clase, lo primero que hace el intérprete de Python es crear un objeto vacío en algún lugar de la memoria. Y justamente después, llama a la función __init__() de la clase. Esta función debemos declararla nosotros mismos dentro del bloque de código de la clase.

```
class Libro():
    def __init__(self):
        print("Estoy creando un nuevo objeto libro")

ejemplar1 = Libro()
```

Escribe el código anterior en el editor de IDLE y ejecútalo. Observa como al invocar a la clase Libro para crear un nuevo objeto aparece el mensaje "Estoy creando un nuevo objeto libro". Esto es debido a que, como acabamos de decir, invocar la clase implica también llamar a la función __init_(). Por tanto se ejecuta su código que, por lo pronto, solo consiste en imprimir por pantalla dicho mensaje.

Llegados a este punto aparecen dos cosas nuevas. En primer lugar hemos definido una función dentro del bloque de la clase. En este caso se trata de una función especial que siempre se ejecuta cuando se invoca a una clase (__init()_). Sin embargo, podemos definir todas las funciones que queramos dentro de una clase. En la terminología de la POO, a estas funciones que pertenecen a una clase se les llama *métodos*. Dentro de poco estudiaremos los métodos con más detalle.

La segunda cosa nueva es el argumento self de la función __init__(). Este argumento es una referencia al objeto que se ha creado al invocar la clase. Veamos esto con más detalle. Cuando invocamos a una clase (como Libro), el intérprete de Python crea un objeto en memoria (ya lo hemos visto en el apartado anterior), pues bien, self es una variable que hace referencia a dicho objeto. De esa manera podemos referirnos al objeto generado por la clase desde dentro de la misma clase. Y esto nos sirve, por ejemplo, para asignar datos característicos a los objetos cuando son creados.

2.2. Atributos de una clase

Los conceptos que modelamos poseen unas características propias que podemos representar como variables asociadas a los objetos de la clase. A estas características se les denominan **atributos**. En el caso de un libro, por ejemplo, algunos atributos serían la signatura, el autor y el título. Pues bien, en el momento de la creación de un objeto, aprovechando la ejecución de la función __init_(), podremos asignar tales atributos al objeto.

```
class Libro():
    def __init__(self):
        self.titulo = "Cien Años de Soledad"
        self.autor = "Gabriel García Márquez"
        self.signatura = "CCS/GGM-1"

ejemplar = Libro()
print(ejemplar.autor)
print(ejemplar.titulo)
print(ejemplar.signatura)
```

Recuerda que el argumento self es una referencia al objeto que se crea al invocar la clase. Usando la **notación punto** se asignan los atributos título, autor y signatura al objeto. Esta notación consiste en separar con un carácter "." el nombre del atributo del nombre del objeto (self, en este caso).

Por ejemplo, la expresión ejemplar.titulo se leería: "el atributo titulo del objeto ejemplar". Y será, como cualquier otra variable, una referencia a un valor. En este caso a un valor de tipo string. Y como a cualquier otra variable podemos asignarle un valor mediante el operador "=":

ejemplar.titulo = "Asterix el Galo"

O extraer su valor sin más que usar la notación punto, como se hace en las tres últimas sentencias print() del código anterior.

Ahora, los objetos creados empiezan a parecerse más a lo que entendemos por un libro. El problema es que se parecen a un único libro, concretamente a "Cien años de soledad " de Gabriel García Márquez, pues todos los objetos que creemos con la clase Libro, tendrán el mismo título, autor y signatura. Obviamente esto no puede quedar así. ¿Cómo lo resolvemos? Pues teniendo en cuenta que __init__() no es ni más ni menos que una función, y que a las funciones les podemos pasar los argumentos que queramos, basta con que pasemos los valores del título, autor y signatura como argumentos y lo usemos para inicializar los atributos.

```
class Libro():
    def __init__(self, titulo, autor, signatura):
        self.titulo = titulo
        self.autor = autor
        self.signatura = signatura

ejemplar1 = Libro("Cien años de soledad", "Gabriel García Marquez", "CAS/GGM-1")
ejemplar2 = Libro("Campos de Castilla", "Antonio Machado", "CC/AM-1")

print(ejemplar1.autor)
print(ejemplar1.titulo)
print(ejemplar1.signatura)
print(ejemplar2.autor)
print(ejemplar2.titulo)
print(ejemplar2.signatura)
```

Ahora ya podemos crear objetos de la clase Libro con los mismos atributos (título, nombre y signatura), pero con distintos valores. La clase Libro ya empieza a tomar forma y a ofrecer la funcionalidad que pretendemos.

Ejercicio 2: ¿Por qué crees que solo pasamos 3 argumentos al invocar la clase, si la nueva versión de la función __init_() tiene 4?

Solución. Ya vimos que self, el primer y obligatorio argumento de la función __init__(), es una referencia al objeto que se crea justo después de invocar la clase. Como obviamente el objeto no existe en el momento en que la invocamos, no podemos pasarlo. Es el interprete de Python el que lo introduce en la función __init_() una vez que lo ha creado. Por eso solo tenemos que pasar los argumentos titulo, autor y signatura.

```
Ejercicio 3: Teniendo en cuenta lo que acabamos de ver. ¿Cual será el resultado del siguiente código?
class Libro():
    def __init__(self):
        self.titulo = ""
        self.autor = ""
        self.signatura = ""


ejemplar1 = Libro()
ejemplar2 = Libro()

ejemplar1.autor = "Gabriel García Márquez"
ejemplar1.titulo = "Cien años de soledad"
ejemplar1.signatura = "CAS/GGM-1"

ejemplar1.autor = "Antonio Machado"
ejemplar1.titulo = "Campos de Castilla"
ejemplar1.signatura = "CC/AM-1"

print(ejemplar1.autor)
print(ejemplar1.titulo)
print(ejemplar1.signatura)
print(ejemplar2.autor)
print(ejemplar2.titulo)
print(ejemplar2.signatura)
```

Solución: El resultado es el mismo que en el código anterior, aunque se llega a él de una manera ligeramente distinta. En este caso no pasamos argumentos adicionales a __init_(), e inicializamos los atributos con cadenas de texto vacías. Entonces, una vez que tenemos las referencias a los objetos recién creados, ejemplar1 y ejemplar2, usamos la notación punto y el operador de asignación “=”, para definir los valores de los atributos que nos interesa en cada objeto.

2.3. Métodos de una clase

Las clases y los objetos no solo nos permiten organizar atributos creando nuevos tipos que representan al concepto que deseamos modelar, también nos ofrecen la posibilidad de capturar su comportamiento.

Pensemos en como se comporta un libro en el ámbito de una biblioteca. El libro puede ser prestado (si está en la sala) y también devuelto (si alguien lo tiene prestado). Son dos acciones que describen el comportamiento de un libro, es decir, cosas que que se pueden hacer con él.

Las acciones que se pueden realizar sobre los objetos de una clase son moldeables mediante funciones pertenecientes a la clase. A estas funciones, en la terminología de la POO, se les llama métodos. Veamos como podemos ampliar nuestra clase Libro para tener en cuenta el préstamo.

Primero añadiremos un nuevo atributo, que llamaremos prestado y que almacenará un valor booleano para indicar si el libro está prestado (True) o está en sala (False).

```
class Libro():
    def __init__(self, titulo, autor, signatura):
        self.titulo = titulo
        self.autor = autor
        self.signatura = signatura
        self.prestado = False
```

Cuando creamos un objeto libro, suponemos que está en la sala, por eso inicializamos el atributo prestado con el valor False.

Ahora vamos a añadir un método (es decir, una función) que llamaremos prestar().

```
class Libro():
    def __init__(self, titulo, autor, signatura):
        self.titulo = titulo
        self.autor = autor
        self.signatura = signatura
        self.prestado = False

    def prestar(self):
        if self.prestado:
            print(f"Lo siento, el libro {self.signatura} ya está prestado")
        else:
            print(f"Se acaba de prestar el libro {self.signatura}")
            self.prestado = True
```

Observa como en la definición de la función prestar(), igual que ocurría en la función especial __init_(), usamos de nuevo el argumento self. Y, de nuevo, representa una referencia al objeto que se ha creado cuando se invocó la clase Libro. Este argumento nos permite acceder a los atributos del objeto desde sus propios métodos para desarrollar su lógica.

Como un libro que ya está prestado no se puede volver a prestar hasta que se devuelva, lo primero que se hace es comprobar si el atributo prestado es True. En ese caso se emite un mensaje indicando que el libro no está disponible para préstamo y no se hace nada más. En caso contrario, se emite un mensaje para notificar que se acaba de prestar el libro y se cambia el valor del atributo prestado a True. Así queda reflejado el nuevo estado del libro en los atributos del propio libro. Y de esta manera hemos mejorado la representación del concepto Libro, no solo mediante atributos característicos si no también añadiendo una faceta de su comportamiento.

Cuando queramos prestar (o más bien, intentar prestar) un objeto libro, llamaremos al método prestar() usando la notación punto:

```
ejemplar1 = Libro("Gabriel García Márquez", "Cien años de soledad", "CAS/GGM-1")
ejemplar1.prestar()
```

Y aquí sucede algo importante con el argumento self. Si en la definición de la función prestar() hemos usado un argumento (self) pero al invocarla no estamos pasando nada dentro de los paréntesis, ¿cómo le llega? Si te ha quedado claro que self es una referencia al objeto, seguramente ya tendrás la respuesta. El método prestar() “sabe” que self no es ni más ni menos que la parte que está a la izquierda del punto, es decir, ejemplar1. Por eso, cuando invocamos cualquier método de un objeto, el primer argumento está implícito en el propio objeto. Así que simplemente no se pasa.

Ejercicio 4: Usando la definición de la clase Libro que acabamos de dar, crea un nuevo libro y ejecuta su método prestar() dos o más veces consecutivas. Explica lo que ocurre.

Solución: El código quedaría así:

```
class Libro():
    def __init__(self, titulo, autor, signatura):
        self.titulo = titulo
        self.autor = autor
        self.signatura = signatura
        self.prestado = False

    def prestar(self):
        if self.prestado:
            print(f"Lo siento, el libro {self.signatura} ya está prestado")
        else:
            print(f"Se acaba de prestar el libro {self.signatura}")
            self.prestado = True



ejemplar1 = Libro("Gabriel García Márquez", "Cien años de soledad", "CAS/GGM-1")

ejemplar1.prestar()
ejemplar1.prestar()
ejemplar1.prestar()
```

Y el resultado sería:

```
Se acaba de prestar el libro CAS/GGM-1
Lo siento, el libro ya está prestado
Lo siento, el libro ya está prestado
```

La primera vez que se llama al método prestar(), como el atributo prestado es False, pues se acaba de crear el objeto, la condición no se cumple y se entra por la rama else, con lo que se emite el mensaje Se acaba de prestar el libro CAS/GGM-1, y se cambia el valor del atributo prestado a True. Las siguientes veces que se llama a prestar(), la condición se cumple y solo se emite el mensaje que indica que el libro ya está prestado.

Ejercicio 5: Ahora queremos tener más control sobre el préstamo, de manera que no solo queremos saber que el libro está prestado, si no a quién está prestado. ¿Cómo modificarías la clase Libro para conseguir este control extra?

Solución: Una posible solución sería la siguiente:

```
class Libro():
    def __init__(self, titulo, autor, signatura):
        self.titulo = titulo
        self.autor = autor
        self.signatura = signatura
        self.prestado = False
        self.prestado_a = None

    def prestar(self, codigo_persona):
        if self.prestado:
            print(f"Lo siento, el libro {self.signatura} ya está prestado")
        else:
            print(f"Se acaba de prestar el libro {self.signatura}")
            self.prestado = True
            self.prestado_a, codigo_persona)
```

Hemos añadido un nuevo atributo prestado_a, que contendrá el código de la persona que tiene el libro, en caso de que se haya prestado, o el valor None si el libro está en la sala. También hemos añadido el argumento codigo_persona al método prestar(), para recoger el código de la persona a la que se ha prestado el libro. Dicho argumento se usará cuando se produzca el préstamo del libro, es decir, cuando se entre por la rama else de la condición.

La forma de invocar el método prestar ahora sería:

ejemplar1.prestar(“786476723B”)

Observa, de nuevo, que el argumento self está implicito en el propio objeto y no se pasa en los argumentos de la función.

Ejercicio 6: Para completar nuestra clase Libro, es evidente que nos falta un método devolver(). Impleméntalo y pruébalo.

Solución: Una posible solución sería la siguiente.

```
class Libro():
    def __init__(self, titulo, autor, signatura):
        self.titulo = titulo
        self.autor = autor
        self.signatura = signatura
        self.prestado = False
        self.prestado_a = None

    def prestar(self, codigo_persona):
        if self.prestado:
            print(f"Lo siento, el libro {self.signatura} ya está prestado")
        else:
            print(f"Se acaba de prestar el libro {self.signatura}")
            self.prestado = True
            self.prestado_a = codigo_persona

    def devolver(self):
        if self.prestado:
            print(f"Libro {self.signatura} devuelto por {self.prestado_a}")
            self.prestado = False
            self.prestado_a = None
        else:
            print(f"El libro {self.signatura} se encuentra en la sala")
```

Se trata simplemente de comprobar si el libro está prestado o no, y de reasignar los atributos prestado y prestado_a adecuadamente, esto es, a False y None respectivamente, en el caso de que estuviera prestado.

2.4. Métodos dunder (o mágicos)

Cualquier objeto creado en Python al invocar una clase, sea la que sea, tiene incorporados unos métodos especiales denominados métodos dunder o mágicos. Se llaman así porque el nombre de todos ellos comienza y termina con dos caracteres underline (__). Ya hemos visto uno de ellos, el más importante de todos: __init__(), pues se llama cada vez que una clase es invocada.

Aunque son muchos los métodos dunder, por su utilidad solo veremos uno más, el método __str__(). Esta función es llamada cuando se requiere una representación en forma de cadena de texto (string) del objeto, por ejemplo cuando se hace un print() del objeto.

Al principio de este módulo usamos el método __str__() sin saberlo. En efecto cuando hicimos:

```
ejemplar1 = Libro()
ejemplar2 = Libro()

print(ejemplar1)
print(ejemplar2)
```

Como la función print() requiere un valor de tipo string para imprimirlo por pantalla, pero ni ejemplar1 ni ejemplar2 son strings, lo que hace el intérprete de Python internamente es llamar a las funciones:

```
print(ejemplar1.__str__())
print(ejemplar2.__str_())
```

Que tienen una implementación por defecto y muestran la clase a la que pertenece el objeto y la posición de memoria que ocupa.

```
<__main__.Libro object at 0x110073ac0>
<__main__.Libro object at 0x1101040d0>
```

Pero podemos cambiar este comportamiento sin más que modificar la definición del método __str__(). El siguiente código muestra una posible redefinición del método mágico __str__() en nuestra clase Libro.

```
class Libro():
    def __init__(self, titulo, autor, signatura):
        self.titulo = titulo
        self.autor = autor
        self.signatura = signatura
        self.prestado = False
        self.prestado_a = None

    def prestar(self, codigo_persona):
        if self.prestado:
            print(f"Lo siento, el libro {self.signatura} ya está prestado")
        else:
            print(f"Se acaba de prestar el libro {self.signatura}")
            self.prestado = True
            self.prestado_a = codigo_persona

    def devolver(self):
        if self.prestado:
            print(f"Libro {self.signatura} devuelto por {self.prestado_a}")
            self.prestado = False
            self.prestado_a = None
        else:
            print(f"El libro {self.signatura} se encuentra en la sala")

    def __str__(self):
        return f"Objeto de tipo libro con signatura {self.signatura}"
```

Se trata, como puedes ver, de añadir la definición de la función __str__() de manera que devuelva la cadena que deseemos.

3. Relaciones entre objetos

Hasta ahora hemos aprendido que la POO ofrece mecanismos para diseñar y crear nuestros propios tipos de datos, que usamos las clases para definir tales tipos y que, a partir de estas clases, creamos objetos a su imagen y semejanza. La idea es modelar conceptos de la vida real que intervienen en los problemas que resolvemos. Pero claro, estos conceptos no son cosas aisladas. Lo normal es que estén relacionados entre sí de alguna manera. Y este es el próximo giro de tuerca en el estudio de la POO: entender como podemos relacionar entre sí los distintos tipos de objetos que vamos creando en nuestros programas.

En esta sección veremos los dos tipos de relaciones más habituales que se pueden dar entre objetos: la composición y la herencia.

3.1. Composición

Sigamos con nuestro ejemplo de biblioteca. Como los libros son prestados y devueltos por personas, en el análisis del problema aparecerán los conceptos Libro y Persona. Y, además se ve que entre ellos hay una relación clara que podemos expresar así: una persona puede tener prestado ninguno, uno o varios libros.

Simplificando la expresión, es como si una persona estuviera compuesta de 0, 1 o varios libros. O en el sentido opuesto, como si un libro estuviera compuesto de una persona. En la jerga de la POO a este tipo de relaciones se les conoce como “has-a relationship” (en inglés, of course).

¿Y como se implementa esto en Python con objetos? Por lo pronto ya tenemos la clase Libro, con lo que podemos crear objetos Libro. Pero, obviamente, antes de poder relacionar los objetos Libro con los objetos Persona, necesitamos una clase Persona con la que poder crear estos últimos.

Pues vamos a ello. El siguiente código define una clase que modela algunos aspectos básicos de una persona:

```
class Persona():
    def __init__(self, nombre, apellidos, nif):
        self.nombre = nombre
        self.apellidos = apellidos
        self.nif = nif
```

Ahora observa lo siguiente: El atributo nombre de los objetos Persona[1] es un objeto de tipo string. Por otro lado una persona tiene un nombre. O también podemos decir que está compuesta por un nombre (entre otras cosas). Por tanto existe una relación de composición entre el objeto nombre (de tipo string)y el objeto persona (de tipo Persona). Entonces la relación de composición se implementa simplemente creando un atributo de una clase en el objeto de la otra clase. Vamos, lo que llevamos haciendo desde que comenzamos el tema.

Supongamos por lo pronto que una persona solo puede tener un libro. Siguiendo la idea de lo que acabamos de contar vamos a montar la relación.

```
class Libro():
    def __init__(self, titulo, autor, signatura):
        self.titulo = titulo
        self.autor = autor
        self.signatura = signatura
        self.prestado = False
        self.prestado_a = None

class Persona():
    def __init__(self, nombre, apellidos, nif):
        self.nombre = nombre
        self.apellidos = apellidos
        self.nif = nif
        self.libro = None
```

Así de fácil, simplemente hemos añadido un atributo libro a los objetos de tipo Persona, que será una referencia a algún objeto Libro cuando la persona haya realizado un préstamo. Por otro lado, el mismo atributo prestado_a que habíamos creado en la clase Persona, podemos reutilizarlo para que en lugar de almacenar un código, almacene una referencia al objeto persona que lo haya tomado prestado.

De esta manera, cuando un objeto esté compuesto por otros objetos, basta con que añadamos estos últimos como atributos del primero. Por otro lado, aunque las relaciones se dan en ambas direcciones, no siempre es necesario expresar las dos partes de la relación. Todo dependerá de como hayamos diseñado nuestro modelo.

Vamos a profundizar en el estudio de la composición con los siguientes ejercicios.

Ejercicio 7:

1. Añade el código necesario a las clases Libro y Persona para que cuando se produzca un préstamo quede reflejado en el objeto Libro qué persona lo tiene y en el objeto Persona qué libro ha tomado prestado.
2. Después crea dos personas y un libro.
3. Entonces presta el libro a la primera persona.
4. Presta el libro a la segunda persona.
5. Usando el objeto Libro, imprime por pantalla quien lo tiene prestado.
6. Devuelve el libro.
7. Presta el libro a la segunda persona.
8. Usando el objeto Libro, imprime por pantalla quien lo tiene prestado.

Solución: El siguiente código resuelve el problema planteado.

```
class Libro():
    def __init__(self, titulo, autor, signatura):
        self.titulo = titulo
        self.autor = autor
        self.signatura = signatura
        self.prestado = False
        self.prestado_a = None

    def prestar(self, persona):
        if self.prestado:
            print(f"Lo siento, el libro {self.signatura} ya está prestado")
        else:
            self.prestado = True
            self.prestado_a = persona
            print(f"Se acaba de prestar el libro {self.signatura} a {self.prestado_a}")

    def devolver(self):
        if self.prestado:
            print(f"Libro {self.signatura} devuelto por {self.prestado_a}")
            self.prestado = False
            self.prestado_a = None

        else:
            print(f"El libro {self.signatura} se encuentra en la sala")

    def __str__(self):
        return f"Objeto de tipo libro con signatura {self.signatura}"


class Persona():
    def __init__(self, nombre, apellidos, nif):
        self.nombre = nombre
        self.apellidos = apellidos
        self.nif = nif
        self.libro = None

    def __str__(self):
        return f"{self.nombre} {self.apellidos}"

persona1 = Persona("Juan", "García", "11111111")
persona2 = Persona("Pepe", "Carmona", "4444444")
libro1 = Libro("Campos de castilla", "Antonio Machado", "CC/AM-1")

# prestamos el libro1 a persona1
libro1.prestar(persona1)
# prestamos el libro2 a persona2
libro1.prestar(persona2)
# ¿quién tiene el libro1?
print(libro1.prestado_a)
# la persona1 devuelve el libro
libro1.devolver()
# prestamos el libro a la persona2
libro1.prestar(persona2)
# ¿quién tiene el libro1?
print(libro1.prestado_a)
```

El código de la clase Libro es prácticamente el mismo que el que escribimos al final de la sección “Clases y objetos”, la única diferencia es que hemos usado el atributo prestado_a para referenciar a objetos Persona, en lugar de para almacenar un código.

Por otro lado, también hemos añadido el método mágico __str__() al objeto Persona. De esta manera, cuando hacemos print(libro1.prestado_a), obtenemos como resultado una representación más amigable del objeto Persona. Si no hubiéramos implementado dicho método, el programa funcionaría también, pero la salida de la sentencia print en lugar de ser así:

```
Pepe Carmona
```

sería algo así:

```
<__main__.Persona object at 0x10cd97df0>
```

lo cual es bastante menos legible.

Ejercicio 8:

1. Modifica el código anterior para hacer posible que una persona pueda tener prestados cualquier cantidad de libros.
2. Crea 2 personas y 3 libros.
3. Presta el libro1 a la persona1
4. Presta el libro2 a la persona1
5. Presta el libro1 a la persona2
6. Presta el libro3 a la persona2
7. Imprime por pantalla los libros que tiene la persona1 y la persona2
8. Devuelve el libro2

9. Presta el libro2 a la persona2
10. Imprime por pantalla los libros que tiene la persona1 y la persona2

Solución:

```
class Libro():
    def __init__(self, titulo, autor, signatura):
        self.titulo = titulo
        self.autor = autor
        self.signatura = signatura
        self.prestado = False
        self.prestado_a = None

    def prestar(self, persona):
        if self.prestado:
            print(f"Lo siento, el libro {self.signatura} ya está prestado")
        else:
            self.prestado = True
            self.prestado_a = persona
            persona.libros.append(self)
            print(f"Se acaba de prestar el libro {self.signatura} a {self.prestado_a}")

    def devolver(self):
        if self.prestado:
            print(f"Libro {self.signatura} devuelto por {self.prestado_a}")
            persona = self.prestado_a
            persona.libros.remove(self)
            self.prestado = False
            self.prestado_a = None
        else:
            print(f"El libro {self.signatura} se encuentra en la sala")

    def __str__(self):
        return f"Objeto de tipo libro con signatura {self.signatura}"

    def __repr__(self):
        return self.__str__()

class Persona():
    def __init__(self, nombre, apellidos, nif):
        self.nombre = nombre
        self.apellidos = apellidos
        self.nif = nif
        self.libros = []

    def __str__(self):
        return f"{self.nombre} {self.apellidos}"


persona1 = Persona("Juan", "García", "11111111")
persona2 = Persona("Pepe", "Carmona", "4444444")
libro1 = Libro("Campos de castilla", "Antonio Machado", "CC/AM-1")
libro2 = Libro("Cien años de soledad", "Gabriel García Márquez", "CAS/GGM-1")
libro3 = Libro("El arte de amar", "Erich Fromm", "EAA/EF-1")

# prestamos el libro1 a persona1
libro1.prestar(persona1)
# prestamos el libro2 a la persona1
libro2.prestar(persona1)
# prestamos el libro1 a persona2
libro1.prestar(persona2)
# prestamos el libro3 a la persona2
libro3.prestar(persona2)
# ¿qué libros tiene la persona 1?
print(persona1.libros)
# ¿qué libros tiene la persona 2?
print(persona2.libros)
# se devuelve el libro2
libro2.devolver()
# prestamos el libro2 a la persona2
libro2.prestar(persona2)
# ¿qué libros tiene la persona 1?
print(persona1.libros)
# ¿qué libros tiene la persona 2?
print(persona2.libros)
```

Los cambios que hemos realizado son los siguientes. En primer lugar, hemos cambiado el nombre del atributo libro por libros, puesto que necesitamos una referencia a una lista de libros, y semánticamente es más correcto usar el plural. Aunque al ordenador esto del plural le da lo mismo, piensa que principalmente escribimos el código para las personas.

Por otro lado, en el método prestar() de la clase Libro, hemos añadido el libro al atributo libros del objeto Persona que se ha pasado como argumento.

```
persona.libros.append(self)
```

Esta expresión merece una explicación. Primero, persona.libros es una lista de Python (List), y las listas son objetos, y como tales tienen métodos;

append() es un método de las listas que sirve para añadir elementos. Por último, ¿qué queremos añadir? Pues el libro que se está prestando, es decir el atributo self que se pasa implícitamente cuando se usa el método prestar() del objeto libro.

Por último, cuando un libro se devuelve, también hay que eliminarlo de la lista libros de la persona que le corresponda. Esto lo hacemos en el método devolver() de la clase Libro:

```
persona.libros.remove(self)
```

La explicación de la expresión anterior es idéntica a la de la expresión usada para añadir libros, solo que ahora usamos el método remove() de las listas para eliminarlo.

1Estrictamente hablando deberíamos decir: “el atributo nombre de los objetos creados con la clase Persona”, pero abusando del lenguaje, porque ya sabemos de que hablamos, acortamos la expresión. Lo de abusar del lenguaje es algo que se hace mucho en el mundo de la programación y, en ocasiones puede causar confusión, especialmente a los que se están iniciando.

3.2. Herencia

El otro tipo de relación que puede darse entre los objetos es el de generalización, más conocido como herencia. Ya sabemos que un objeto tiene un tipo asignado que se corresponde con el de la clase que se usó para crearlo. Pero lo interesante es que también puede pertenecer a otras clases más generales.

Esto, que posiblemente te haya sonado extraño, seguramente que lo vas a entender con un ejemplo de la vida real (al fin y al cabo la POO trata de hacer modelos computacionales de la realidad). Piensa en el concepto de Persona y en el de Perro. Ambos son cosas distintas con sus propias características. Sin embargo comparten algo con un concepto más genérico, el de Animal. En efecto, una persona **es** un animal y un perro **es** un animal. En la terminología de la POO, a este tipo de relación se le conoce como “is-a relationship”.

Pues bien, esta idea de que hay conceptos (más específicos) que **son** a su vez otros conceptos (más generales), es lo que se trata de resolver con el mecanismo de herencia en la POO. Para explicar como se definen las relaciones de herencia en Python, vamos a modelar una familia de figuras planas cuyos tamaños podemos ampliar o reducir (escalar). Concretamente modelaremos círculos y cuadrados.

Podemos caracterizar a una figura plana cualquiera con tres atributos: el nombre, el color y el área. Siguiendo lo que hemos aprendido hasta el momento, la siguiente podría ser una clase que modela el concepto general FiguraPlana.

```
class FiguraPlana():
    def __init__(self, nombre="", color="", area=None):
        self.nombre = nombre
        self.color = color
        self.area = area

    def escalar(self, cantidad):
        self.area = self.area*cantidad*2
```

Fíjate que en la definición del método __init__() hemos usado parámetros por defecto. Al fin y al cabo un método no es más que una función, así que podemos usar todo lo que has aprendido sobre las funciones. La ventaja de esto es que podemos crear objetos de la clase FiguraPlana pasando los valores de atributos que conozcamos en el momento de crear el objeto. Aquellos que no pasemos se inicializarán con los valores por defecto. Además podemos pasar los argumentos indicando su nombre, como muestra el siguiente ejemplo:

```
fig = FiguraPlana(color="rojo")
```

Por otro lado, el método escalar() utiliza el hecho de que al multiplicar cada dimensión de una figura plana por un factor, el área de la figura se multiplica por ese factor elevado al cuadrado.

Ejercicio 9: Con esta clase podemos modelar cualquier figura plana. Escribe el código para crear, usando la clase FiguraPlana que acabamos de proponer, un cuadrado verde de lado 6 y un círculo rojo de radio 4.

Solución: El siguiente código muestra una posible solución.

```
lado = 6
cuadrado = FiguraPlana("cuadrado", "verde", lado*lado)

radio = 4
PI = 3.1416
circulo = FiguraPlana("círculo", "rojo", PI*radio*radio)
```

Observa que para definir el área de cada figura hemos tenido que usar su fórmula, en el caso del cuadrado area = lado x lado y en el del círculo area = PI x radio^2. Lo cual está bien, pero es poco operativo. Una solución más adecuada pasaría por definir una clase para crear cuadrados y otra para crear círculos.

La cosa es que ambas clases también tendrían los mismos atributos y métodos que la clase FiguraPlana. Aunque, además, el círculo añadiría el radio como atributo y el cuadrado necesitaría añadir el atributo lado. Y es aquí donde entra la herencia: podemos reutilizar la clase FiguraPlana para crear las clases Cuadrado y Circulo. De manera que todo lo que tiene la clase FiguraPlana, también lo tengan Cuadrado y Circulo. ¿Y cómo hacemos esto? Pues así de fácil.

```
class Cuadrado(FiguraPlana):
    pass

class Circulo(FiguraPlana):
    pass
```

Sin más que pasar la clase FiguraPlana como argumento de las clases Cuadrado y Circulo, estas últimas han **heredado** todos los atributos y métodos de la primera. Decimos que la clase Cuadrado y la clase Circulo han heredado de la clase FiguraPlana. O también que la clase Circulo (o Cuadrado) es hija de la clase FiguraPlana. O también que la clase FiguraPlana es padre de las clase Circulo y Cuadrado. O también que la clase Cuadrado (o Circulo) son más específicas que la clase FiguraPlana. O también que la clase FiguraPlana generaliza a las clases Circulo y Cuadrado. En fin, muchas formas de decir lo mismo. Lo importante es que un objeto creado con la clase Circulo (o Cuadrado), es un Circulo (o un Cuadrado) y también una FiguraPlana.

Ejercicio 10: Repite el ejercicio anterior usando las clases Cuadrado y Circulo.

Solución: Esta sería la solución:

```
lado = 6
cuadrado = Cuadrado("cuadrado", "verde", lado*lado)

radio = 4
PI = 3.1416
circulo = Circulo("círculo", "rojo", PI*radio*radio)
```

Prueba a usar algunos atributos y métodos de la clase FiguraPlana, ya verás como funciona perfectamente.

Y además podemos saber el tipo específico del objeto usando la función **type()**:

```
print(type(circulo))
print(type(cuadrado))
```

Aunque el hecho de poder usar type() para discriminar entre objetos ya es una ventaja, aún no hemos exprimido las posibilidades que nos ofrece la herencia. En efecto, lo bueno de heredar es que te puedes quedar con lo que quieras, modificar lo que te convenga y/o añadir lo que necesites a las nuevas clases que heredan de la clase padre.

En nuestro caso necesitamos un atributo extra en ambas clases hijas (el lado y el radio) y cambiar la forma en que se escalan, pues habría que tener en cuenta también el cambio que se produce en la longitud del lado o del radio cuando se realiza la operación de escalado.

Además, podemos usar las fórmulas para calcular el área del círculo y del cuadrado en el momento en que creamos los objetos. El siguiente código realiza todas estas modificaciones:

```
class Cuadrado(FiguraPlana):
    def __init__(self, nombre="", color="", lado=None):
        self.lado = lado
        area = lado*lado if lado else None
        super().__init__(nombre, color, area)


    def escalar(self, cantidad):
        super().escalar(cantidad)
           self.lado = cantidad*self.lado


class Circulo(FiguraPlana):
    def __init__(self, nombre="", color="", radio=None, area=None):
        self.pi = 3.1416
        self.radio = radio
        area = self.pi*radio*radio if radio else None
        super().__init__(nombre, color, area)

   def escalar(self, cantidad):
        super().escalar(cantidad)
           self.radio = cantidad*self.radio
```

La novedad en este código es el uso de la función super(), la cual devuelve una referencia a la clase padre, es decir, a FiguraPlana. Además hemos cambiado los argumentos de la función __init__() tanto en la clase Cuadrado como en la Circulo. En la primera hemos añadido el argumento lado, para inicializar el atributo self.lado, y en la segunda hemos añadido el argumento radio, para inicializar el atributo self.radio. Después, en ambos casos, hemos calculado el área con la fórmula correspondiente y acto seguido hemos llamado a la función __init__() de la clase padre, para aprovechar su código y que se inicialicen los atributos comunes.

Por otro lado, como el escalado de un cuadrado implica un cambio en la longitud del lado proporcional a la cantidad por la que se escala y lo mismo ocurre con un círculo pero aplicándolo al radio, hemos modificado el método escalar() en ambas clases. De nuevo llamamos a la función super() para realizar la modificación del área, que es común a ambas clases, y acto seguido modificamos el lado o el radio, según el caso.

Ejercicio 11: Escribe el código de una clase que modelen el concepto de triángulo.

Solución: El siguiente código muestra una posible solución.

```
class Triangulo(FiguraPlana):
    def __init__(self, nombre="", color="", a=None, b=None, c=None):
        self.a = a
        self.b = b
        self.c = c
        area = self.calcula_area()
        super().__init__(nombre=nombre, color=color, area=area)

    def calcula_area(self):
        import math
        a = self.a
        b = self.b
        c = self.c
        p = (a+b+c)/2
        area = math.sqrt(p*(p-a)*(p-b)*(p-c))

        return area

    def escalar(self, cantidad):
        self.escalar(cantidad)
        self.a = cantidad*self.a
        self.b = cantidad*self.b
        self.c = cantidad*self.c
```

En este código, como el cálculo del área de un triángulo dado sus lados es un poco farragosa, hemos decidido crear un método específico, que hemos llamado calcular_area(), para calcular el área. Por lo demás es similar al de las clases Cuadrado y Circulo.

3.3. Conclusión

La Programación Orientada a Objetos (POO) se basa en la idea de agrupar en clases las características comunes de los conceptos reales que modelamos cuando programamos. A partir de estas clases, que actúan como plantillas, podemos crear objetos del mismo tipo pero que mantienen su identidad. Los problemas que se resuelven para un objeto determinado, se resuelven de la misma forma para otro objeto del mismo tipo, con lo que reutilizamos el mismo código. Además, los objetos pueden relacionarse entre sí mediante relaciones de herencia y composición, con lo que se enriquece enormemente la capacidad de modelar sistemas reales que pueden tener bastante complejidad. En definitiva, la POO nos facilita la organización y reutilización del código.

Para finalizar, aquí tienes un videotutorial que repasa los conceptos fundamentales que hemos visto, aplicados para programar una granja, que se compone de animales. En la granja habrá diferentes tipos de animales, así que se crea una clase Animal, y luego varias clases de animales (Vaca, Perro, Cerdo, Gallina) que heredan todas de Animal. La clase Granja tiene un atributo que es una lista de objetos, que son instancias de la clase Animal, haciendo uso de la composición.