

# PROGRAMACION



# ORIENTADA A OBJETOS

## ¿Qué es la Programación Orientada a Objetos?

La **Programación Orientada a Objetos** (POO) es un **paradigma de programación**, es decir, un modelo o un estilo de programación que nos da unas guías sobre cómo trabajar con él. Se basa en el **concepto de clases y objetos**. Este tipo de programación se utiliza para estructurar un programa de software en piezas simples y reutilizables de planos de código (clases) para crear instancias individuales de objetos.

A lo largo de la historia, han ido apareciendo diferentes paradigmas de programación. Lenguajes secuenciales como COBOL o procedimentales como Basic o C, se centraban más en la lógica que en los datos. Otros más modernos como Java, C# y Python, utilizan paradigmas para definir los programas, siendo la Programación Orientada a Objetos la más popular.

Con el paradigma de **Programación Orientado a Objetos** lo que buscamos es dejar de centrarnos en la lógica pura de los programas, para empezar a pensar en objetos, lo que constituye la base de este paradigma. Esto nos ayuda muchísimo en sistemas grandes, ya que, en vez de pensar en funciones, pensamos en las relaciones o interacciones de los diferentes componentes del sistema.

Un programador diseña un programa de software organizando piezas de información y comportamientos relacionados en una plantilla llamada clase. Luego, se crean objetos individuales a partir de la plantilla de clase. Todo el programa de software se ejecuta haciendo que varios objetos interactúen entre sí para crear un programa más grande.



# ¿Por qué POO?

La Programación Orientada a objetos permite que el código sea reutilizable, organizado y fácil de mantener. Sigue el principio de desarrollo de software utilizado por muchos programadores **DRY (Don't Repeat Yourself)**, para evitar duplicar el código y crear de esta manera programas eficientes. Además, evita el acceso no deseado a los datos o la exposición de código propietario mediante la encapsulación y la abstracción, de la que hablaremos en detalle más adelante.

## Motivación de la programación orientada a objetos

Durante años, los programadores se han dedicado a construir aplicaciones muy parecidas que resolvían una y otra vez los mismos problemas. Para conseguir que los esfuerzos de los programadores puedan ser reutilizados se creó la posibilidad de utilizar módulos. El primer módulo existente fue la función, que somos capaces de escribir una vez e invocar cualquier número de veces.

Sin embargo, la función se centra mucho en aportar una funcionalidad dada, pero no tiene tanto interés con los datos. Es cierto que la función puede recibir datos como parámetros y puede devolverlos, pero los trata como una estructura muy volátil, centrada en las operaciones. Simplemente hace su trabajo, procesando los parámetros recibidos y devuelve una respuesta.

En las aplicaciones en realidad los datos están muy ligados a la funcionalidad. Por ejemplo, podemos imaginar un punto que se mueve por la pantalla. El punto tiene unas coordenadas y podemos trasladarlo de una posición a otra, sumando o restando valores a sus coordenadas. Antes de la programación orientada a objetos ocurría que cada coordenada del punto tenía que guardarse en una variable diferente (dos variables para ser exacto: x, y) y las funciones de traslación estaban almacenadas por otra parte. Esta situación no facilitaba la organización del código ni tampoco su reutilización.

Con la Programación Orientada a Objetos se buscaba resolver estas situaciones, creando unas mejores condiciones para poder desarrollar aplicaciones cada vez más complejas, sin que el código se volviera un caos. Además, se pretendía dar una de pautas para realizar las cosas de manera que otras personas puedan utilizarlas y adelantar su trabajo, lo que deriva en mayores facilidades para la reutilización del código.

La POO no es difícil, pero es una manera especial de pensar, a veces subjetiva de quien la programa, de manera que la forma de hacer las cosas puede ser diferente según el programador. Aunque podamos hacer los programas de formas distintas, no todas ellas son correctas, lo difícil no es programar orientado a objetos sino programar bien. Y programar bien es fundamental porque así podemos aprovechar de todas las ventajas de la POO.

## Cómo se piensa en objetos

Pensar en términos de objetos es muy parecido a cómo lo haríamos en la vida real. Por ejemplo, vamos a pensar en un coche para tratar de modelizarlo en un esquema de POO.

Diríamos que el coche es el elemento principal que tiene una serie de características, como podrían ser el color, el modelo o la marca. Además, tiene una serie de funcionalidades asociadas, como pueden ser ponerse en marcha, parar o aparcar.



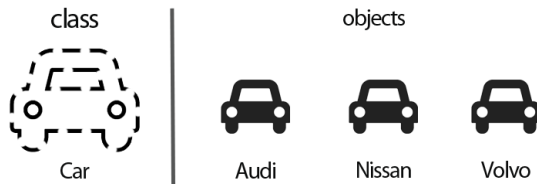
Por tanto, pensar en objetos requiere analizar qué elementos vas a manejar en tus programas, tratando de identificar sus características y funcionalidades. Una vez tengamos un ecosistema de objetos, éstos colaborarán entre sí para resolver los objetivos de las aplicaciones.

*Quizás al principio puede ser un poco complejo dar ese salto, para pensar en objetos, pero con el tiempo será una tarea que realizarás automáticamente.*

## Clases, objetos e instancias

### Manejando el concepto de clase

En un esquema de programación orientada a objetos "el coche" sería lo que se conoce como "Clase". La clase contiene la definición de las características de un modelo (el coche), como el color o la marca, junto con la implantación de sus funcionalidades, como arrancar o parar.



Las características definidas en la clase las llamamos propiedades y las funcionalidades asociadas las llamamos métodos.

Para entender este concepto tan importante dentro de la Programación Orientada a Objetos, podemos pensar que la clase es como un libro, que describe como son todos los objetos de un mismo tipo. La clase coche describe cómo son todos los coches del mundo, es decir, qué propiedades tienen y qué funcionalidades deben poder realizar y, por supuesto, cómo se realizan.

### Manejando el concepto de objetos

A partir de una clase podemos crear cualquier número de objetos de esa clase. Por ejemplo, a partir de la clase "el coche" podemos crear un coche rojo que es de la marca Ford y modelo Fiesta, otro verde que es de la marca Seat y modelo Ibiza.

Por tanto, los objetos son ejemplares de una clase, o elementos concretos creados a partir de una clase. Puedes entender a la clase como el molde y a los objetos como concreciones creadas a partir del molde de clase.

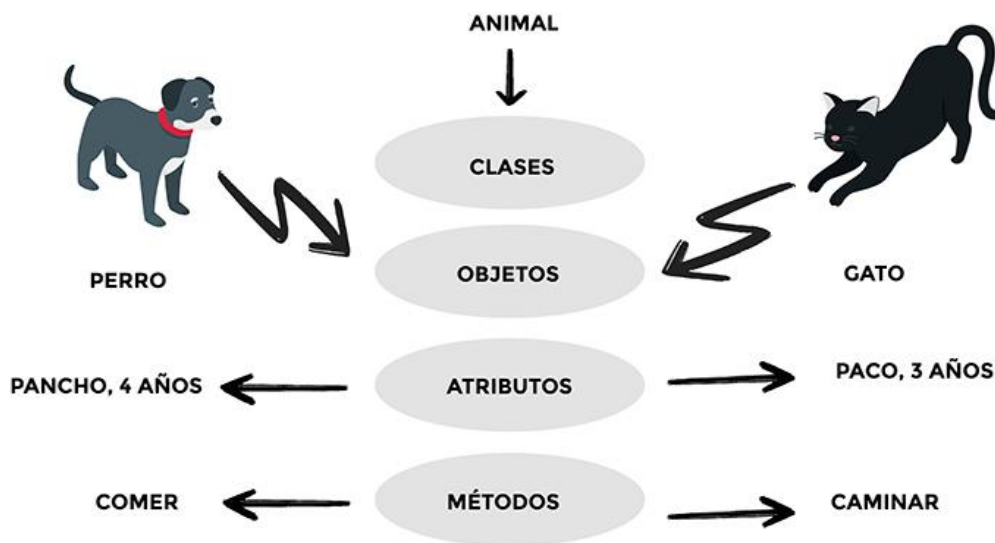
Entonces, ¿Cómo se crean los programas orientados a objetos? Resumiendo, mucho, consistiría en hacer clases y crear objetos a partir de estas clases. Las clases forman el modelo a partir del que se estructuran los datos y los comportamientos.

Que te quede claro que el primer y más importante concepto de la POO es la **distinción entre clase y objeto**.

Recordemos, Una **clase** es una plantilla. Define de manera genérica cómo van a ser los objetos de un determinado tipo. Por ejemplo, una clase para representar a animales puede llamarse 'animal' y tener una serie de **atributos**, como 'nombre' o 'edad' (que normalmente son propiedades), y una serie con los comportamientos que estos pueden tener, como caminar o comer, y que a su vez se implementan como métodos de la clase (funciones).

## Otros ejemplos concretos de clases y objetos

Un ejemplo sencillo de un objeto, como decíamos antes, podría ser un animal. Un animal tiene una edad, por lo que creamos un nuevo atributo de 'edad' y, además, puede envejecer, por lo que definimos un nuevo método. Datos y lógica. Esto es lo que se define en muchos programas como la definición de una clase, que es la definición global y genérica de muchos objetos.



Con la clase se pueden crear instancias de un objeto, cada uno de ellos con sus atributos definidos de forma independiente. Con esto podríamos crear un gato llamado *Paco*, con 3 años de edad, y otro animal, este tipo perro y llamado *Pancho*, con una de edad de 4 años. Los dos están **definidos por la clase animal**, pero son dos instancias distintas. Por lo tanto, llamar a sus métodos puede tener resultados diferentes. Los dos comparten la lógica, pero cada uno tiene su estado de forma independiente.

Estos son simplemente ejemplos simples de clases, que también nos sirven para destacar la reutilización que podemos conseguir con los objetos. La clase coordenada la podremos usar en infinidad de programas de gráficos, mapas, etc. Por su parte, la clase coche la podrás utilizar en un programa de gestión de un taller de coches o en un programa de gestión de un parking. A partir de clase coche y crearán diversos objetos de tipo coche para hacer las operativas tanto en la aplicación del taller como en la del parking.

Todo esto, junto con los principios que vamos a ver a continuación, son herramientas que nos pueden ayudar a escribir un código mejor, más limpio y reutilizable.

## Los objetos colaboran entre sí

En los lenguajes puramente orientados a objetos, tendremos únicamente clases y objetos. Las clases permitirán definir un número indeterminado de objetos, que colaboran entre ellos para resolver los problemas.

Con muchos objetos de diferentes clases conseguiremos realizar las acciones que se desean implementar en la funcionalidad de la aplicación.

Además, las propias aplicaciones como un todo, también serán definidas por medio de clases. Es decir, el taller de coches será una clase, de la que podremos crear el objeto taller de coches, que utilizará objetos coche, objetos de clase herramienta, objetos de clase mecánico, objetos de clase recambio, etc.

## **Clases en Programación Orientada a Objetos**

Como habrás podido entender, las clases son declaraciones de objetos, también se podrían definir como abstracciones de objetos. Esto quiere decir que la definición de un objeto es la clase. Cuando programamos un objeto y definimos sus características y funcionalidades en realidad lo que estamos haciendo es programar una clase. En los ejemplos anteriores en realidad hablábamos de las clases coche o fracción porque sólo estuvimos definiendo, aunque por encima, sus formas.

### **Propiedades en clases**

Las propiedades o atributos son las características de los objetos o ejemplares de una clase. Cuando definimos una propiedad normalmente especificamos su nombre y su tipo.

Aunque no es una manera muy correcta de hablar, nos podemos hacer a la idea de que las propiedades son algo así como variables donde almacenaremos los datos relacionados con los objetos.

### **Métodos en las clases**

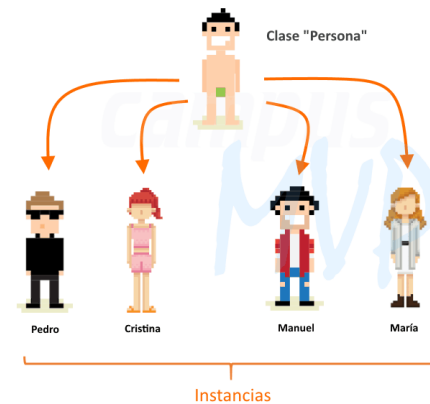
Son las funcionalidades asociadas a los objetos, que son implantadas o programadas dentro de las clases. Es decir, cuando estamos programando las clases, las funciones que creamos dentro asociadas a esas clases las llamamos métodos.

Aunque los métodos son como funciones, es importante que los llames métodos para que quede claro que son funciones que existen dentro del contexto de una clase, funciones que podremos invocar sobre todos los objetos creados a partir de una clase.

## **Objetos en Programación Orientada a Objetos**

Los objetos son ejemplares de una clase. A partir de una clase puedo crear ejemplares (objetos) de esa clase, que tendrán por tanto las características y funcionalidades definidas en esa clase.

Cuando creamos un ejemplar tenemos que especificar la clase a partir de la cual se creará. Esta acción de crear un objeto a partir de una clase se llama instanciar (que viene de una mala traducción de la palabra instace que en inglés significa ejemplar). Por ejemplo, un objeto de la clase fracción es por ejemplo  $\frac{3}{5}$ . El concepto o definición de fracción sería la clase, pero cuando ya estamos hablando de una fracción en concreto  $\frac{4}{7}$ ,  $\frac{8}{1000}$ , o cualquier otra, la llamamos objeto.



Para crear un objeto se tiene que escribir una instrucción especial que puede ser distinta dependiendo el lenguaje de programación que se emplee, pero será algo parecido a esto.

**miCoche = new Coche()**

Con la palabra "new" especificamos que se tiene que crear una instancia de la clase que sigue a continuación.

En este caso "Coche" sería el nombre de la clase. Con la palabra "new" se crea una nueva instancia de coche, un objeto concreto de la clase coche y ese objeto se almacena en la variable "miCoche".

Dentro de los paréntesis podríamos colocar parámetros con los que inicializar el objeto de la clase coche, como podría ser su color, o su marca.

## Estados en objetos

Cuando tenemos un objeto sus propiedades toman valores. Por ejemplo, cuando tenemos un coche la propiedad color tomará un valor en concreto, como por ejemplo rojo o gris metalizado. El valor concreto de una propiedad de un objeto se llama estado.

Para acceder a un estado de un objeto para ver su valor o cambiarlo se utiliza el operador punto.

`miCoche.color = "rojo"`

El objeto es miCoche, luego colocamos el operador punto y por último el nombre de la propiedad a la que deseamos acceder. En este ejemplo estamos cambiando el estado de la propiedad "color" del objeto al valor "rojo". Esto lo hacemos con una simple asignación.

## Mensajes en objetos (métodos)

Un mensaje en un objeto es la acción de efectuar una llamada a un **método**. Por ejemplo, cuando le decimos a un objeto coche que se ponga en marcha estamos pasándole el mensaje "ponte en marcha".

Para mandar mensajes a los objetos utilizamos el operador punto, seguido del método que deseamos invocar y los paréntesis, como en las llamadas a las funciones.

**miCoche.ponteEnMarcha()**

En este ejemplo pasamos el mensaje "ponteEnMarcha" al objeto "miCoche".

Después del nombre del método que queremos invocar hay que colocar paréntesis, igual que cuando invocamos una función. Dentro de los paréntesis irían los parámetros, si es que el método los requiere.

Para continuar vamos a definir de manera más formal los conceptos que acabamos de introducir anteriormente.

## Pilares de la Programación Orientada a Objetos

Existen muchos conceptos en programación orientada a objetos, como clases y objetos, sin embargo, en el desarrollo de software con programación orientada a objetos, existen un conjunto de ideas fundamentales que forman los cimientos del desarrollo de software. A estos 4 conceptos que vamos a ver les llamamos los 4 pilares de la programación orientada a objetos.



Esto no quiere decir que fuera de estos 4 pilares no existan otras ideas igual de importantes, sin embargo, estos 4 pilares representan la base de ideas más avanzadas, por lo que es crucial entenderlos.

Estos pilares son: abstracción, encapsulamiento, herencia y polimorfismo.

### La encapsulación

La encapsulación contiene **toda la información importante de un objeto de ntro del mismo** y solo expone la información seleccionada al mundo exterior.

Esta propiedad permite asegurar que la información de un objeto esté oculta para el mundo exterior, agrupando en una Clase las características o atributos que cuentan con un acceso privado, y los comportamientos o métodos que presentan un acceso público.

La encapsulación de cada objeto es responsable de su propia información y de su propio estado. La única forma en la que este se puede modificar es mediante los propios métodos del objeto. Por lo tanto, los atributos internos de un objeto deberían ser **inaccesibles desde fuera**, pudiéndose modificar sólo llamando a las funciones correspondientes. Con esto conseguimos mantener el estado a salvo de usos indebidos o que puedan resultar inesperados.

Usamos de ejemplo un coche para explicar la encapsulación. El coche comparte información pública a través de las luces de freno o intermitentes para indicar los giros (interfaz pública). Por el contrario, tenemos la interfaz interna, que sería el mecanismo propulsor del coche, que está oculto bajo el capó. Cuando se conduce un automóvil es necesario indicar a otros conductores tus movimientos, pero no exponer datos privados sobre el tipo de carburante o la temperatura del motor, ya que son muchos datos, lo que confundiría al resto de conductores.

### La abstracción

La abstracción es cuando **el usuario interactúa solo con los atributos y métodos seleccionados de un objeto**, utilizando herramientas simplificadas de alto nivel para acceder a un objeto complejo.

En la programación orientada a objetos, los programas suelen ser muy grandes y los objetos se comunican mucho entre sí. El concepto de abstracción **facilita el mantenimiento de un código de gran tamaño**, donde a lo largo del tiempo pueden surgir diferentes cambios.

Así, la abstracción se basa en usar **cosas simples para representar la complejidad**. Los objetos y las clases representan código subyacente, ocultando los detalles complejos al usuario. Por consiguiente, supone una extensión de la encapsulación. Siguiendo con el ejemplo del coche, no es necesario que conozcas todos los detalles de cómo funciona el motor para poder conducirlo.

## La herencia

La herencia define **relaciones jerárquicas entre clases**, de forma que atributos y métodos comunes puedan ser reutilizados. Las clases principales extienden atributos y comportamientos a las clases secundarias. A través de la definición en una clase de los atributos y comportamientos básicos, se pueden crear clases secundarias, ampliando así la funcionalidad de la clase principal y agregando atributos y comportamientos adicionales.

Volviendo al ejemplo de los animales, se puede usar una sola clase de animal y agregar un atributo de tipo de animal que especifique el tipo de animal. Los diferentes tipos de animales necesitarán diferentes métodos, por ejemplo, las aves deben poder poner huevos y los peces, nadan. Incluso cuando los animales tienen un método en común, como moverse, la implementación necesitaría muchas declaraciones «sí» para garantizar el comportamiento de movimiento correcto. Por ejemplo, las ranas saltan, mientras que las serpientes se deslizan. El principio de herencia nos permite solucionar este problema.

## El polimorfismo

El polimorfismo consiste en **diseñar objetos para compartir comportamientos**, lo que nos permite procesar objetos de diferentes maneras. Es la capacidad de presentar la misma interfaz para diferentes formas subyacentes o tipos de datos. Al utilizar la herencia, los objetos pueden anular los comportamientos principales compartidos, con comportamientos secundarios específicos. El polimorfismo permite que el mismo método ejecute diferentes comportamientos de dos formas: anulación de método y sobrecarga de método.

Alrededor de estos principios de la programación orientada a objetos se construyen muchas cosas. Por ejemplo, los Principios SOLID, o los Patrones de diseño, que son recetas que se aplican a problemas recurrentes que se han encontrado y se repiten en varios proyectos.

## Beneficios de Programación Orientada a Objetos

- **Reutilización** del código.
- Convierte cosas complejas en **estructuras simples reproducibles**.
- Evita la **duplicación de código**.
- Permite **trabajar en equipo** gracias al encapsulamiento ya que minimiza la posibilidad de duplicar funciones cuando varias personas trabajan sobre un mismo objeto al mismo tiempo.
- Al estar la clase bien estructurada permite la **corrección de errores** en varios lugares del código.
- **Protege la información** a través de la encapsulación, ya que solo se puede acceder a los datos del objeto a través de propiedades y métodos privados.
- La abstracción nos permite **construir sistemas más complejos** y de una forma más sencilla y organizada.



# Críticas a la programación orientada a objetos

El modelo de programación orientada a objetos ha sido criticado por los desarrolladores por múltiples razones. La mayor preocupación es que la programación orientada a objetos hace demasiado hincapié en el componente de datos del desarrollo de software y no se centra lo suficiente en la computación o los algoritmos. Además, el código OOP puede ser más complicado de escribir y tardar más en compilarse.

Los métodos alternativos a la programación orientada a objetos incluyen:

- programación funcional
- programación estructurada
- programación imperativa
- Los lenguajes de programación más avanzados brindan a los desarrolladores la opción de combinar estos modelos.



## Conclusión

La Programación Orientada a Objetos es actualmente el **paradigma** que más se utiliza para diseñar aplicaciones y programas informáticos. Son muchas sus ventajas, principalmente cuando necesitas resolver desafíos de programación complejos. Permite una mejor estructura de datos y reutilización del código, lo que facilita el ahorro de tiempo a largo plazo. Eso sí, para ello se requiere pensar bien en la estructura del programa, planificar al comienzo de la codificación, así como analizar los requisitos en clases simples y reutilizables que se pueden usar para diseñar instancias de objetos.

# POO EN PYTHON

A continuación pensemos en el **comportamiento** de estas clases. Preguntémonos qué cosas hacen, qué tipo de acciones pueden realizar.

Por ejemplo, un auto puede *arrancar, detenerse, girar a la izquierda, acelerar, frenar, encender* sus luces. Un semáforo puede *cambiar de estado*. Un bar puede *abrir, cerrar, servirte una cerveza, cobrarla, modificar la lista de precios*. Una persona puede *hablar, dormir, conducir* un coche, *tomarse una cerveza* en un bar.

En terminología de la programación orientada a objetos, a estas funciones que determinan el *comportamiento* de una clase se las conoce como **métodos**.

Empleando un lenguaje puramente pythonista, tanto a los atributos como a los métodos aquí descritos se les conoce como simplemente atributos. Para diferenciar unos de otros, Python emplea el término **atributos de datos** para referirse a los datos característicos de la clase. No obstante, al menos mientras continuemos presentando los conceptos básicos, seguiremos distinguiendo aquí entre atributos y métodos.

Ya sabemos qué es una **clase**, y que está compuesta de **atributos** y **métodos**. Nuestra labor ahora consistirá en **modelar** en Python estas clases.

*Modelar* no es otra cosa sino crear una **abstracción** que represente de algún modo una determinada realidad.

Para crear en Python la clase *Coche* comenzamos usando la palabra reservada **class**, seguida del nombre de la clase y del símbolo de dos puntos:

```
class Coche:
    instrucción 1
    instrucción 2
    ...
    instrucción n
```

La clase más simple de todas no haría nada:

```
class Coche:
    pass
```

La sentencia **pass** pasa completamente, no hace absolutamente nada. Es útil en contextos en los que se requiere la presencia de al menos una instrucción.

Típicamente, indentados en la definición, introduciremos los *atributos* y *métodos* de que consta la clase *Coche*.

El nombre *Coche* lo escribimos con la primera letra en mayúsculas, pues es buena práctica entre la POO que los nombres de clases empiecen así.

Introduzcamos algunos atributos:

```
class Coche:
    marca = ''
    modelo = ''
    color = ''
    numero_de_puertas = 0
    cuenta_kilometros = 0
    velocidad = 0
    arrancado = False
```

Hemos utilizado, provisionalmente, valores iniciales para los atributos: la cadena vacía para los tipo string, cero para los numéricos y False para el booleano, pero podrían haber sido otros cualesquiera. En el momento en que comencemos a fabricar objetos de esta clase todos estos atributos podrán recibir su valor concreto y particular.

Definamos ahora los métodos de la clase *Coche*:

```
class Coche:
    marca = ''
    modelo = ''
    color = ''
    numero_de_puertas = 0
    cuenta_kilometros = 0
    velocidad = 0
    arrancado = False

    def arrancar(self):
        pass

    def parar(self):
        pass

    def acelerar(self):
        pass

    def frenar(self):
        pass

    def pitar(self):
        pass

    def consultar_velocimetro(self):
        pass

    def consultar_cuenta_kilometros(self):
        pass
```

Observa que los métodos se implementan exactamente igual que las funciones. Como tales, junto a la palabra clave **def** aparece el nombre del método seguido de unos paréntesis con la declaración de parámetros formales, finalizando con dos puntos e indentando el cuerpo del método a continuación. Al igual que las funciones ordinarias, los métodos podrán o no devolver un valor.

Fíjate en **self**, un parámetro que **debe figurar siempre al comienzo de todo método** y hace **referencia al objeto concreto sobre el que se está invocando el método**.

Como no hemos decidido que código colocar en los métodos, lo rellenamos con la palabra `pass`, para no obtener errores del compilador al estar vacíos.

Ahora escribamos algo de código en los métodos de nuestra clase.

```
def arrancar(self):
    if not self.arrancado:
        print('Roarrrr')
        self.arrancado = True
    else:
        # Dale al encendido estando el coche arrancado y escucha...
        print('Kriiiiiiiiiicccc')

def parar(self):
    self.arrancado = False

def acelerar(self):
    if self.arrancado:
        self.velocidad = self.velocidad + 1

def frenar(self):
    if self.velocidad > 0:
        self.velocidad = self.velocidad - 1

def pitar(self):
    print('Bip Bip Bip')

def consultar_velocimetro(self):
    return self.velocidad

def consultar_cuenta_kilometros(self):
    return self.cuenta_kilometros
```

La siguiente podría ser una primera aproximación simple al modelado de una persona:

```
class Persona:

    sexo = ''
    nombre = ''
    edad = 0
    coche = Coche() # El coche que conduce esa persona
```

```
def saludar(self):  
    print('Hola, me llamo', self.nombre)  
  
def dormir(self):  
    print('Zzzzzzzzzz')  
  
def obtener_edad(self):  
    return self.edad
```

Fíjate en que uno de los atributos es precisamente de la clase *Coche* que acabamos de definir:

```
coche = Coche() # El coche que conduce esa persona
```

No confundas el nombre del atributo (*coche*, en minúsculas), con el nombre de la clase (*Coche*, con la primera en mayúsculas). Aquí podemos observar que entre los atributos de una clase, además de tipos comunes (números, strings, listas, etc...), pueden figurar objetos de cualquier clase, incluso de las creadas por nosotros mismos.

En la definición de clase nos encontramos sus características o atributos, que toman forma de variables (marca, modelo, color, etc...) y la funcionalidad propia de la clase o métodos, que nos indican qué cosas podemos hacer con ella, representados mediante funciones (arrancar(), acelerar(), frenar(), etc.).

Pero esto no es más que un esquema, un molde. Lo que queremos ahora es fabricar coches concretos, como un Seat León de cinco puertas o un Ford Fiesta de tres.

Al hecho de «fabricar» objetos a partir de un molde de clase se le denomina instanciar, y al producto obtenido se le conoce como instancias o, simplemente, objetos de esa clase.

De modo que vamos a asegurarnos que tenemos los conceptos claros: las clases son los moldes a los que recurrimos después para crear objetos concretos.

Para mayor comodidad, supongamos que hemos preparado ya todo el código con la definición de la clase *Coche* y lo hemos incluido en un fichero tal como *coche.py*.

Comprobemos su existencia:

```
>>> Coche
```

```
<class '__main__.Coche'>
```

Para fabricar coches concretos necesitamos invocar a lo que se conoce como el constructor de clase, que se ocupará de crear el objeto en memoria. El rito para que haga acto de presencia es muy simple: basta con escribir el nombre de la clase seguido de un par de paréntesis.

Construyamos un primer coche:

```
>>> coche1 = Coche()
```

Típicamente los paréntesis los utilizaremos para definir valores iniciales del objeto. Más adelante, cuando presentemos el inicializador, aprenderás a usar esa característica.

Parémonos un instante a contemplar nuestra primera fabricación:

```
>>> coche1
```

```
<__main__.Coche object at 0x0234B7D0>
```

Lo que nos está diciendo el intérprete es que coche1 es un objeto de tipo Coche y está ubicado en la dirección de memoria 0x0234B7D0.

Cada objeto posee en Python un identificador único que podemos consultar con la función id():

```
>>> id(coche1)
```

```
37009360
```

Ahora, probemos a fabricar un segundo automóvil:

```
>>> coche2 = Coche()
```

Aunque de la misma clase, se trata en efecto de un objeto diferente:

```
>>> id(coche2)
```

```
37634800
```

Para acceder a los atributos y métodos del objeto recurrimos a la notación punto, separando el nombre del objeto del atributo o método mediante un punto.

Establezcamos algunos atributos de coche1:

```
coche1.marca = "Seat"  
coche1.modelo = "León"  
coche1.color = "negro"
```

Y otros tantos de coche2:

```
coche2.marca = "Ford"  
coche2.modelo = "Fiesta"  
coche2.numero_de_puertas = 3
```

Esta forma de proceder, que puede parecer natural, no es la común. Tal como ya hemos apuntado, para dar valores iniciales a los objetos suele emplearse la figura del inicializador o constructor, como veremos más adelante. Lo importante ahora es comprender bien el concepto.

Comprobemos los atributos de coche1:

```
>>> coche1.marca  
'Seat'  
>>> coche1.modelo  
'León'  
>>> coche1.color  
'negro'  
>>> coche1.numero_de_puertas  
0  
>>> coche1.cuenta_kilometros  
0  
>>> coche1.velocidad  
0  
>>> coche1.arrancado  
False
```

Aprecia como los atributos que no hemos inicializado expresamente toman los valores que tenían en la definición de la clase.

Juguemos ahora un poco con los métodos, funciones que definen lo que podemos hacer con los objetos.

Traigamos a la palestra el primero de ellos. Lo reescribo aquí por comodidad:

```
def arrancar(self):
    if not self.arrancado:
        print('Roarrrr')
        self.arrancado = True
    else:
        # Dale al encendido estando el coche arrancado y escucha...
        print('Kriiiiiiiiiicccc')
```

El parámetro self, tal como explicamos, es obligatorio y debe figurar en primero en la declaración del método. Hace referencia al objeto en cuestión sobre el que se aplicará el método.

Para invocar el método, recurrimos nuevamente a la notación punto, pasando entre paréntesis los argumentos que requiere el método en particular sin contar a self. Como en arrancar sólo existe self, a efectos prácticos es como si no tuviera ninguno:

```
>>> coche1.arrancar()
```

```
Roarrrr
```

Aunque no esté presente en la invocación, se está pasando implícitamente como self el nombre del método que llama a arrancar(), en este caso, coche1. Observa el código del método. Lo primero que hace es comprobar que el coche no está arrancado, en cuyo caso imprime «Roarrrr», cambiando a continuación el atributo arrancado a True. Verifiquemos que en efecto se ha realizado esto último:

```
>>> coche1.arrancado
```

```
True
```

Naturalmente, coche2 permanece completamente ajeno a estas operaciones, pues arrancar no ha actuado sobre él, sino sobre coche1. Comprobemos que sigue detenido:

```
>>> coche2.arrancado
```

```
False
```

Estando coche1 arrancado, si intentamos arrancarlo de nuevo obtendremos un chirriante y desagradable ruido propio de forzar el motor de arranque sin necesidad:

```
>>> coche1.arrancar()
```



Kriiiiiiiiiicccc

Revisemos algunos métodos más de la definición de la clase Coche:

```
def parar(self):
    self.arrancado = False

def acelerar(self):
    if self.arrancado:
        self.velocidad = self.velocidad + 1

def frenar(self):
    if self.velocidad > 0:
        self.velocidad = self.velocidad - 1
```

Experimentemos con ellos. Cada vez que aceleramos, la velocidad del vehículo se incrementa en una unidad:

```
>>> coche1.acelerar()
```

```
>>> coche1.velocidad
```

1

```
>>> coche1.acelerar()
```

```
>>> coche1.velocidad
```

2

Para detener el vehículo, primero lo frenamos completamente:

```
>>> coche1.frenar()
```

```
>>> coche1.frenar() # Al tener velocidad 2 aplicamos dos veces el método
```

```
>>> coche1.velocidad
```

0

Y, a continuación, sacamos la llave de contacto:

```
>>> coche1.parar()
```

```
>>> coche1.arrancado
```

```
False
```

Ahora veamos cómo usar el inicializador o constructor de la clase.

Vimos que para fabricar coches a partir de este esquema, invocamos al constructor:

```
>>> coche1 = Coche()
```

Como puedes comprobar, el valor de los atributos de datos del objeto recién creado son los que aparecen en la definición de la clase. Por ejemplo: Para marca obtendremos vacío, pues así lo definimos hasta ahora.

```
>>> coche1.marca
```

```
''
```

De modo que, para poder montarte en el coche, lo primero que habría que hacer es darle algo de forma y color:

```
>>> coche1.marca = "Seat"
```

```
>>> coche1.modelo = "Ibiza"
```

```
>>> coche1.color = "Blanco"
```

```
>>> coche1.numero_de_puertas = 3
```

Los demás campos podemos dejarlos con sus valores iniciales, Lo que quiero recalcar aquí, es que esta manera de proceder, que habría que repetir con cada coche que fabricáramos, no es, obviamente, ni práctica, ni elegante ni inteligente, ni una buena practica.

¿Por qué no facilitamos estos valores iniciales al crear la instancia del objeto?, para ello usamos los constructores.

Es decir, algo así como:

```
coche1 = Coche("Seat", "Ibiza", "Blanco", 3)
```

Mucho más elegante, menos propenso a errores y, como veremos, permitiéndonos además un mayor control sobre los valores que pueden tomar los atributos.

Para poder comunicarnos de este modo con la clase Coche necesitamos ampliar su definición incorporando un nuevo método: el inicializador o constructor

El inicializador es un método particular cuya función es obvia: ocuparse de los rudimentos iniciales necesarios a la hora de construir el objeto. Toma la forma siguiente, obsérvala con atención:

```
def __init__(self, argumento1, argumento2, ..., argumentoN):  
    código.....
```

La palabra `init` va precedida y seguida por dos símbolos de subrayado (dos delante y dos detrás). Esto, que puede parecer algo esotérico, es una buena convención práctica de Python para evitar posibles conflictos de nombres.

Observa que el primer argumento es `self`, como en cualquier otro método, que ya sabes que referencia al objeto actual. Los demás argumentos deberán cuadrar con la invocación en el constructor:

```
coche1 = coche(argumento1, argumento2, ..., argumentoN)
```

Recuerda que `self` no se indica dentro del constructor.

Estamos en condiciones de introducir código en el inicializador que, en este caso, es bien simple:

```
def __init__(self, marca, modelo, color, puertas):  
    self.marca = marca  
    self.modelo = modelo  
    self.color = color  
    self.numero_de_puertas = puertas
```

En la declaración de parámetros he empleado `puertas` en lugar de `numero_de_puertas` simplemente para ilustrar y recordar la flexibilidad que tenemos a la hora de elegir nombres de parámetros en las funciones. Naturalmente, en la última línea del código hay que utilizar el nombre real del atributo para que éste se nutra del valor facilitado.

La clase Coche quedaría entonces así:

```
class Coche:

    marca = ''
    modelo = ''
    color = ''
    numero_de_puertas = 0
    cuenta_kilometros = 0
    velocidad = 0
    arrancado = False

    def __init__(self, marca, modelo, color, puertas):
        self.marca = marca
        self.modelo = modelo
        self.color = color
        self.numero_de_puertas = puertas

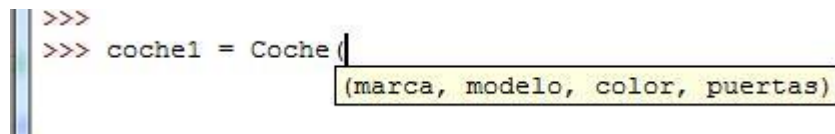
    def arrancar(self):
        if not self.arrancado:
            print('Roarrrr')
            self.arrancado = True
        else:
            # Dale al encendido estando el coche arrancado y escucha...
            print('Kriiiiiiiiiicccc')

    etcétera, ...
```

En breve la mejoraremos aún más, pero vayamos asegurando los conceptos paso a paso.

Con esta definición, la siguiente llamada al constructor ya es lícita:

```
>>> coche1 = Coche("Seat", "Ibiza", "Blanco", 3)
```



```
>>>
>>> coche1 = Coche(
    (marca, modelo, color, puertas)
```

Los IDEs (en la imagen, IDLE) nos facilitan la vida al decirnos que atributos forman parte del constructor.

Lo que sucede a continuación es que los argumentos «Seat», «Ibiza», «Blanco» y 3 son facilitados, respectivamente, a marca, modelo, color y puertas de `__init__`.

Una vez dentro de `__init__`, `self` referencia al objeto actual que se crea al invocar al constructor, es decir, `coche1`. De modo que interiormente `__init__` realiza esto:

```
coche1.marca = "Seat"
coche1.modelo = "Ibiza"
```

```
coche1.color = "Blanco"
coche1.numero_de_puertas = 3
```

Puedes comprobar que coche1 ha sido inicializado correctamente. Por ejemplo:

```
>>> coche1.color

'Blanco'

>>> coche1.numero_de_puertas

3
```

Ahora sí, nuestra clase Coche empieza a tener clase de verdad. Pero aún podemos hacerla más fina...

Sabemos que las variables en Python no necesitan declararse y que se construyen en el momento en que reciben un valor. Este hecho nos permite simplificar nuestra clase. Si los atributos de datos van a ser inicializados dentro de `__init__`, ¿por qué molestarnos siquiera en dar valores iniciales al definir la clase? Simplifiquemos, entonces:

```
class Coche:

    cuenta_kilometros = 0
    velocidad = 0
    arrancado = False

    def __init__(self, marca, modelo, color, puertas):
        self.marca = marca
        self.modelo = modelo
        self.color = color
        self.numero_de_puertas = puertas

    def arrancar(self):
        if not self.arrancado:
            print('Roarrrr')
            self.arrancado = True
        else:
            # Dale al encendido estando el coche arrancado y escucha...
            print('Kriiiiiiiiiicccc')

    etcétera, ...
```

Si revisan detenidamente, hemos dejado sólo los atributos que expresamente no establecemos en `__init__`.

Aún podemos ir más allá. ¿Por qué no incluir toda la inicialización en `__init__`, incluso la de aquellos miembros que no se faciliten explícitamente en la invocación al constructor?

```
class Coche:

    def __init__(self, marca, modelo, color, puertas):
        self.marca = marca
        self.modelo = modelo
        self.color = color
        self.numero_de_puertas = puertas
        self.cuenta_kilometros = 0
        self.velocidad = 0
        self.arrancado = False

    def arrancar(self):
        if not self.arrancado:
            print('Roarrrr')
            self.arrancado = True
        else:
            # Dale al encendido estando el coche arrancado y escucha...
            print('Krriiiiiiiiiicccc')

etcétera, etcétera...
```

Los atributos de datos, objetos también en sí (en Python, todo lo es), serán construidos en el momento de la inicialización.

Por lo general, bien sea a la hora de inicializar un objeto, o bien llegado el momento de modificar cualquiera de sus atributos de datos, es buena práctica dejar que sean los métodos quienes se encarguen de ello, en lugar de atacar a las variables miembro directamente. Dentro de un método podemos analizar el valor facilitado, comprobar que sea legítimo, corregirlo o incluso rechazarlo si no procede.

No sería muy sensato escribir código como el siguiente:

```
coche1.velocidad = 1000
```

No hay coche que pueda alcanzar 1000 Km/h. Tampoco podría estar parado y, en el instante siguiente, circular a 100 Km/h. Usando los métodos acelerar y frenar que hemos definido en la clase se garantiza un incremento o decremento de la velocidad gradual. Estos métodos, además, podrían asegurarse de que no se excediera en ningún caso un valor máximo por mucho que pisáramos el acelerador (ni mínimo cuando frenásemos, algo que ya hace el método frenar).

Tienen ya las bases suficientes para empezar a utilizar la Programación Orientada a Objetos en sus programas. La idea es que una vez escriban su primer programa POO, no quieran volver a la programación estructurada tradicional.

# EJERCICIOS

## 1. Clase Cuenta

Crea una clase llamada Cuenta que tendrá los siguientes atributos: titular y cantidad.

- El titular será obligatorio y la cantidad es opcional.
- Crea un constructor que cumpla lo anterior.
- Crear atributos

Tendrá dos métodos especiales:

- Ingresar: se ingresa una cantidad a la cuenta, si la cantidad introducida es negativa, no se hará nada.
- Retirar: se retira una cantidad a la cuenta, si restando la cantidad actual a la que nos pasan es negativa, la cantidad de la cuenta pasa a ser 0.

## 2. Haz una clase llamada Persona que siga las siguientes condiciones:

- Sus atributos son: nombre, edad, CC, sexo (H hombre, M mujer), peso y altura.
- Por defecto, todos los atributos menos el CC serán valores por defecto según su tipo (0 números, cadena vacía para String, etc.).
- Sexo será femenino por defecto.
- Un constructor con valores por defecto.

Los métodos que se implementaran son:

- calcularIMC(): calcula si la persona está en su peso ideal (peso en kg/(altura<sup>2</sup> en m)), si esta fórmula devuelve un valor menor que 20, la función devuelve un -1, si devuelve un número entre 20 y 25 (incluidos), significa que está por debajo de su peso ideal la función devuelve un 0 y si devuelve un valor mayor que 25 significa que tiene sobrepeso, la función devuelve un 1.
- esMayorDeEdad(): indica si es mayor de edad, devuelve un booleano.
- toString(): devuelve toda la información (formateada en un texto) del objeto.
- generaCC(): genera un número aleatorio de 8 cifras, Este método será invocado cuando se construya el objeto.

Ahora, crea un programa que haga lo siguiente:

- Pide por teclado el nombre, la edad, sexo, peso y altura.
- Crea 3 objetos de la clase anterior, el primer objeto obtendrá las anteriores variables pedidas por teclado, el segundo objeto obtendrá todos los anteriores menos el peso y la altura y el último con los valores por defecto.
- Para cada objeto, deberá comprobar si está en su peso ideal, tiene sobrepeso o por debajo de su peso ideal con un mensaje.

- Indicar para cada objeto si es mayor de edad.
- Por último, mostrar la información de cada objeto.

### 3. Haz una clase llamada Password que siga las siguientes condiciones:

- Que tenga los atributos longitud y contraseña. Por defecto, la longitud será de 8.

Los constructores serán los siguientes:

- Un constructor con la longitud que nosotros le pasemos. Generará una contraseña aleatoria con esa longitud.

Los métodos que implementa serán:

- esFuerte(): devuelve un booleano si es fuerte o no, para que sea fuerte debe tener más de 2 mayúsculas, más de 1 minúscula y más de 5 números.
- generarPassword(): genera la contraseña del objeto con la longitud que tenga.

Ahora, crea un programa que:

- Crea un array de Passwords con el tamaño que tu le indiques por teclado.
- Crea un bucle que cree un objeto para cada posición del array.
- Indica también por teclado la longitud de los Passwords (antes de bucle).
- Crea otro array de booleanos donde se almacene si el password del array de Password es o no fuerte (usa el bucle anterior).
- Al final, muestra la contraseña y si es o no fuerte (usa el bucle anterior). Usa este simple formato:

contraseña1 valor\_booleano1

contraseña2 valor\_booleano2

### 4. Crearemos una supeclase llamada Electrodoméstico con las siguientes características:

- Sus atributos son precio base, color, consumo energético (letras entre A y F) y peso.
- Por defecto, el color será blanco, el consumo energético será F, el precioBase es de 100 € y el peso de 5 kg.
- Los colores disponibles son blanco, negro, rojo, azul y gris.

Los constructores que se implementaran serán

- Un constructor por defecto.

Los métodos que implementara serán:



- `comprobarConsumoEnergetico`: comprueba que la letra es correcta, sino es correcta usara la letra por defecto. Se invocara al crear el objeto.
- `comprobarColor`: comprueba que el color es correcto, sino lo es usa el color por defecto. Se invocara al crear el objeto
- `precioFinal()`: según el consumo energético, aumentara su precio, y según su tamaño, también. Esta es la lista de precios:

Letra	Precio
A	100 €
B	80 €
C	60 €
D	50 €
E	30 €
F	10 €

Tamaño	Precio
Entre 0 y 19 kg	10 €
Entre 20 y 49 kg	50 €
Entre 50 y 79 kg	80 €
Mayor que 80 kg	100 €

Crearemos una subclase llamada Lavadora con las siguientes características:

- Su atributo es carga, además de los atributos heredados.
- Por defecto, la carga es de 5 kg.

Los constructores que se implementaran serán:

- Un constructor por defecto.

Los métodos que se implementara serán:

- `precioFinal()`: si tiene una carga mayor de 30 kg, aumentara el precio 50 €, sino es así no se incrementara el precio. Llama al método padre y añade el código necesario. Recuerda que las condiciones que hemos visto en la clase Electrodoméstico también deben afectar al precio.

Crearemos una subclase llamada Televisión con las siguientes características:

- Sus atributos son resolución (en pulgadas) y sintonizador TDT (booleano), además de los atributos heredados.
- Por defecto, la resolución será de 20 pulgadas y el sintonizador será false.

Los constructores que se implementaran serán:

- Un constructor por defecto.

Los métodos que se implementara serán:

- `precioFinal()`: si tiene una resolución mayor de 40 pulgadas, se incrementara el precio un 30% y si tiene un sintonizador TDT incorporado, aumentara 50 €. Recuerda que las condiciones que hemos visto en la clase `Electrodomestico` también deben afectar al precio.

Ahora crea un programa que realice lo siguiente:

- Crea un array de `Electrodomesticos` de 10 posiciones.
- Asigna a cada posición un objeto de las clases anteriores con los valores que desees.
- Ahora, recorre este array y ejecuta el método `precioFinal()`.
- Deberás mostrar el precio de cada clase, es decir, el precio de todas las televisiones por un lado, el de las lavadoras por otro y la suma de los `Electrodomesticos` (puedes crear objetos `Electrodomestico`, pero recuerda que `Televisión` y `Lavadora` también son `electrodomesticos`).
- Por ejemplo, si tenemos un `Electrodomestico` con un precio final de 300, una lavadora de 200 y una televisión de 500, el resultado final será de 1000 ( $300+200+500$ ) para `electrodomesticos`, 200 para lavadora y 500 para televisión.

## 5. Batalla Superheroe vs Villano

Ahora a pensar!!!!: Deben programar una batalla automática entre 1 superheroe y un villano. La batalla durara hasta que uno de los 2 se quede sin puntos de vida. El daño infligido de un contrincante a otro debe ser aleatorio entre 100 y 200 puntos. Si desean, pueden inventar o adicionarle más cosas (poderes, escudos, etc)