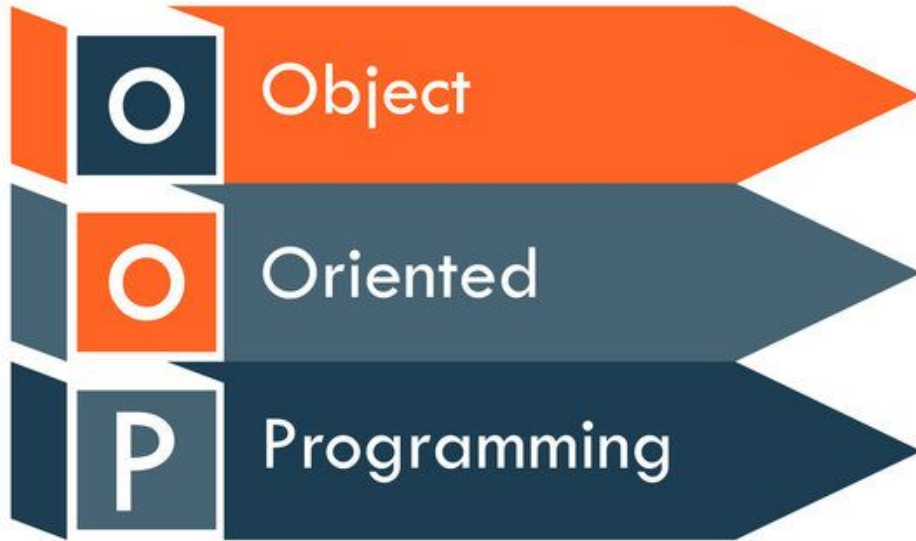


Tema 4: Introducción de Programación Orientada a Objetos



**Conceptos
teóricos**

Cambio de paradigma

- El mundo del desarrollo de software es un mundo en constante evolución y cambio, y allá por los años 60 se empezó a hablar de un nuevo paradigma de desarrollo que era la programación orientada a objetos. El objetivo de la programación orientada a objetos no tenía otro objetivo que no fuera intentar paliar las deficiencias existentes en la programación en ese momento, que eran las siguientes:

Cambio de paradigma

- **Distinta abstracción del mundo:** la programación en ese momento se centraba en comportamientos representado por verbos normalmente, mientras que la programación orientada a objetos se centra en seres, representados por sustantivos normalmente. Se pasa de utilizar funciones que representan verbos, a utilizar clases, que representan sustantivos.
- **Dificultad de modificación y actualización:** los datos suelen ser compartidos por los programas, por lo que cualquier ligera modificación de los datos podría provocar que otro programa dejara de funcionar de forma indirecta.

Cambio de paradigma

- **Dificultad de mantenimiento:** la corrección de errores que existía en ese momento era bastante costosa y difícil de realizar.
- **Dificultad de reutilización:** las funciones/rutinas suelen ser muy dependientes del contexto en el que se crearon y eso dificulta reaprovecharlas en nuevos programas.

Cambio de paradigma

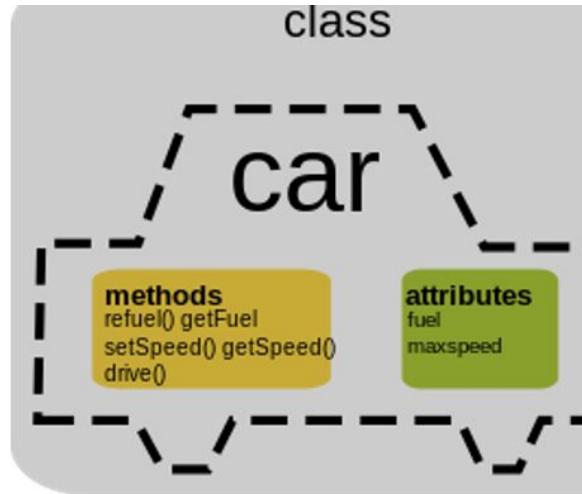
- La programación orientada a objetos, básicamente, apareció para aportar lo siguiente:
 - Nueva abstracción del mundo centrándolo en seres y no en verbos mediante nuevos conceptos como clase y objeto que veremos a continuación.
 - Control de acceso a los datos mediante encapsulación de éstos en las clases.
 - Nuevas funcionalidades de desarrollo para clases, como por ejemplo herencia y composición, que permiten simplificar el desarrollo.

Concepto de clase y objeto

- El nuevo paradigma de programación orientada a objetos está basado en una abstracción del mundo real que nos va a permitir desarrollar programas de forma más cercana a cómo vemos el mundo, pensando en objetos que tenemos delante y acciones que podemos hacer con ellos.
- Una clase es un tipo de dato cuyas variables se llaman objetos o instancias. Es decir, la clase es la definición del concepto del mundo real y los objetos o instancias son el propio “objeto” del mundo real.

Concepto de clase y objeto

- Piensa por un segundo en un coche, antes de ser fabricado un coche tiene que ser definido, tiene que tener una plantilla que especifique sus componentes y lo que puede hacer, pues esa plantilla es lo que se conoce como Clase.
- Una vez el coche es construido, ese coche sería un objeto o instancia de la clase Coche, que es quien define qué es un coche y qué se puede hacer con un coche.

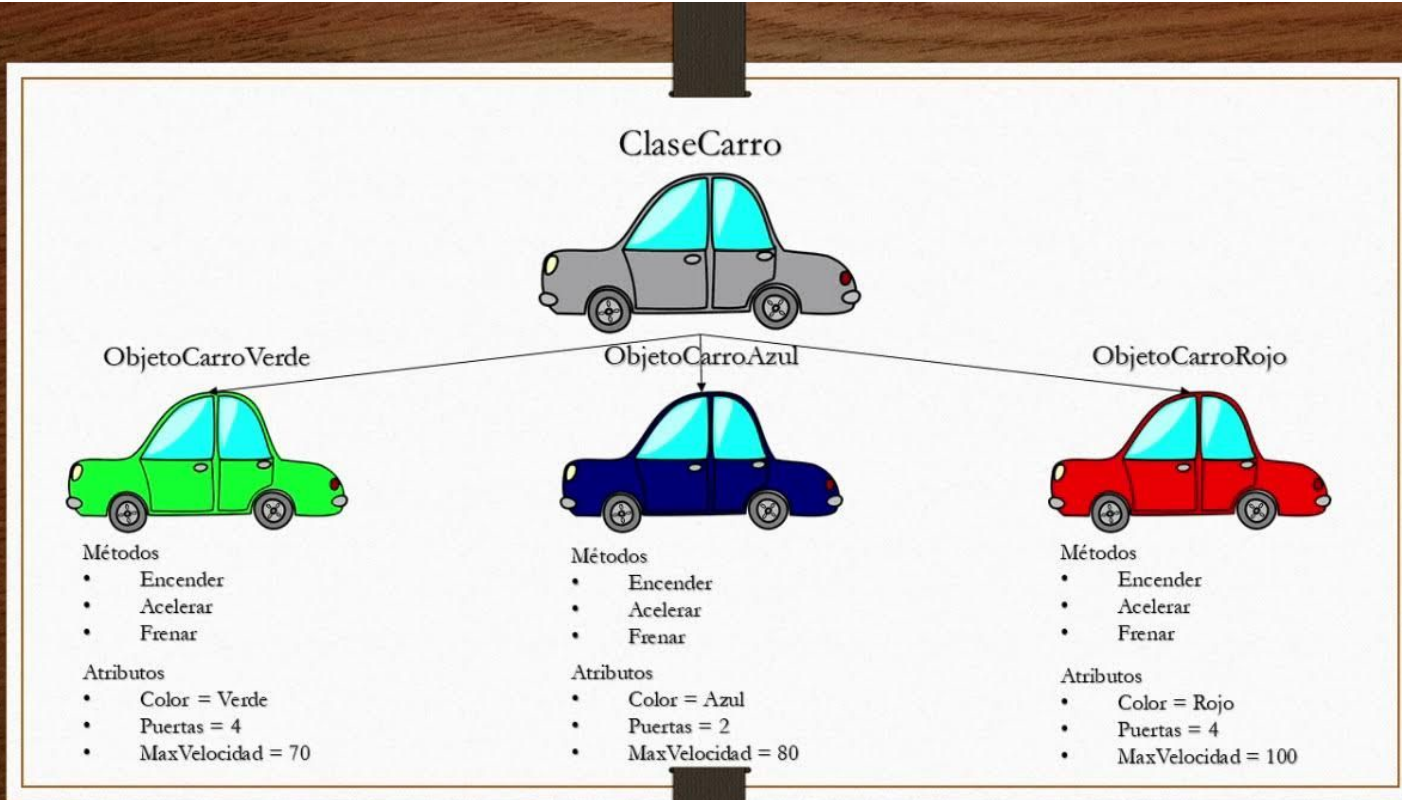


Concepto de clase y objeto

- Las clases están compuestas por dos elementos:
 - **Atributos:** información que almacena la clase.
 - **Métodos:** operaciones que pueden realizarse con la clase.

Piensa ahora en el coche de antes, la clase coche podría tener atributos tales como número de marchas, número de asientos, color... y podría realizar las operaciones tales como subir marcha, bajar marcha, acelerar, frenar, encender el intermitente... Un objeto es un modelo de coche concreto.

Concepto de clase y objeto



Composición

- La composición consiste en la creación de nuevas clases a partir de otras clases ya existentes que actúan como elementos compositores de la nueva. Las clases existentes serán atributos de la nueva clase. La composición te va a permitir reutilizar código fuente.
- Cuando hablemos de composición tienes que pensar que entre las dos clases existe una relación del tipo “tiene un”.
- Por ejemplo, una biblioteca tiene libros, por lo que usamos composición al hacer que la clase *Biblioteca* contenga objetos de la clase *Libro* como sus atributos.

Creación de Clase simple

Clase simple

- En todas las clases que vayamos a crear, es necesario siempre crear un método llamado “__init__”. Es lo que se conoce como el constructor de la clase o inicializador. Es posible incluir parámetros, la forma de especificarlos es igual que se hace en las funciones.
- El método “__init__” es lo primero que se ejecuta cuando creas un objeto de una clase.

Empezaremos creando una clase que represente un punto, con su coordenada X y su coordenada Y. Además, la clase tendrá un método para mostrar la información que poseen ambos puntos. El ejercicio consistirá en la creación de un objeto o instancia de la clase, estableciendo las coordenadas y utilizando el método para mostrar la información de la coordenada.

Clase simple

- Empezamos haciendo el constructor (es decir, la función `__init__`). El código fuente sería el siguiente:

```
class Punto:
```

```
    def __init__ (self, x, y):
```

```
        self.X = x
```

```
        self.Y = y
```

Como vemos, lo primero que hacemos es definir la clase con la palabra **class** y asignándole un nombre a esta. Después definimos la función `__init__`, pasándole los diferentes parámetros.

Clase simple

- No es obligatorio pasar siempre un parámetro por cada atributo, un atributo podría tener un valor por defecto, sin embargo, sí que es recomendable ya que podemos definir lo que se conoce como un **atributo de clase**, es decir, un atributo que será común para todos los puntos. Por ejemplo, si estamos trabajando en 2 dimensiones un atributo común sería que la dimensión $Z = 0$. Quedándose el código de la siguiente manera:

- *class Punto:*

- Z = 0*

- def __init__ (self, x, y):*

- self.X = x*

- self.Y = y*

Clase simple

- Podemos crear una instancia u objeto de nuestra clase pasandole el valor de los atributos. Podemos además usar `type()` donde podemos ver cómo efectivamente el objeto es de la clase Punto.

- *class Punto:*

- Z = 0*

- def __init__ (self, x, y):*

- self.X = x*

- self.Y = y*

mi_punto = Punto(5,3)

print(type(mi_punto)) // <class '__main__.Punto'>

Clase simple

- Podemos acceder a los atributos usando el objeto creado, un "." y el nombre del atributo:
 - `mi_punto = Punto(5,3)`
 - `print(mi_punto.X)` `// 5`
 - `print(mi_punto.Y)` `// 3`
- También podemos acceder al atributo de clase desde el objeto o incluso desde la propia clase (ya que no es necesario crear un objeto para acceder a él).
 - `print(mi_punto.Z)` `// 0`
 - `print(Punto.Z)` `// 0`

Clase simple

- A continuación, vamos a definir el método que mostrará la información del punto. Seguiríamos nuestro código de la siguiente manera:

```
class Punto:  
    Z = 0  
    def __init__ (self, x, y):  
        self.X = x  
        self.Y = y  
    def MostrarPunto(self):  
        print(f"El punto es ({self.X}, {self.Y})")
```

Podríamos ahora mostrar nuestro punto creado:

```
mi_punto.MostrarPunto() // El punto es (5, 3)
```

Clase simple

❖ Para hacer en clase:

- *Añadid más métodos a la clase, como por ejemplo:*
 - *Mover (método para cambiar las coordenadas del punto)*
 - *estaEnElOrigen (método que devolverá True o False según si está en el origen o no)*
 - *reflejar (método que devuelve un nuevo punto reflejado respecto de los ejes)*
 - *distanciaEuclidiana (método que recibe otro punto como parámetro y calcula la distancia euclidiana entre ambos)*
- *Probad a crear más instancias u objetos y probad estos métodos.*

Composición

Composición

- La composición es un concepto de diseño orientado a objetos que modela una relación. Permite crear objetos complejos agrupando objetos más simples.
- Permite reutilizar el código, reduciendo la duplicación y los errores.
- En Python, la composición te permite crear objetos complejos combinando objetos más simples. Esto se hace creando una instancia de una clase (el objeto compuesto) que incluye instancias de otras clases (los componentes). Estos componentes se consideran parte del objeto compuesto y su ciclo de vida lo gestiona el objeto compuesto.

Composición

- Para comprender mejor la composición, consideremos un ejemplo. Supongamos que estamos construyendo un sistema para modelar un coche. Un coche se compone de varias partes como motor, transmisión, ruedas, etc. Cada una de estas partes puede representarse mediante un objeto. Por tanto, podemos utilizar la composición para modelar un coche como un objeto formado por varios componentes más pequeños:

claseCoche:

def __init__(yo):

self.motor = Motor()

self.puertas = Puertas()

self.ruedas = Ruedas()

Composición

- La composición también permite cambiar la implementación de una clase sin afectar las clases que la utilizan. Por ejemplo, si decidimos cambiar la forma en la que funciona el motor, solo se necesita cambiar la clase Motor. No es necesario cambiar las clases que utilizan la clase Coche.

Implementación de composición

- Teniendo la clase que hemos estado utilizando anteriormente (la clase Punto), definiremos una nueva clase llamada Triángulo, que contendrá tres atributos Punto.
- Tal y como te hemos visto, no hay que especificar el tipo de dato de los atributos, por lo que creando tres atributos es suficiente.

Class Triangulo:

```
def __init__ (self, v1,v2,v3):
```

```
    self.V1 = v1
```

```
    self.V2 = v2
```

```
    self.V3 = v3
```


Implementación de composición

- Por ejemplo, podemos crear un método que llame a la función `mostrarPuntos()`, aprovechando el método de la clase `Punto()`:

Class Triangulo:

```
def __init__ (self, v1,v2,v3):
```

```
    self.V1 = v1
```

```
    self.V2 = v2
```

```
    self.V3 = v3
```

```
def MostrarVertices(self):
```

```
    self.V1.MostrarPunto()
```

```
    self.V2.MostrarPunto()
```

```
    self.V3.MostrarPunto()
```

Implementación de composición

- Para probar el código, escribimos lo siguiente:

```
v1 = Punto(3,4)
```

```
v2 = Punto(6,8)
```

```
v3 = Punto(9,2)
```

```
triangulo = Triangulo(v1,v2,v3)
```

```
triangulo.MostrarVertices()
```

- Tendremos la siguiente salida:

```
El punto es ( 3 , 4 )  
El punto es ( 6 , 8 )  
El punto es ( 9 , 2 )  
>>>
```

Implementación de composición

- ❖ Para hacer en clase. Implementar los siguientes métodos en la clase Triángulo:
 - **distanciaEntreTodosLosPuntos():** Calculará la distancia euclidiana entre cada uno de los puntos y las devolverá en una lista.
 - **Perímetro():** calculará el perímetro del triángulo y lo devolverá
 - **tipoTriangulo():** Indicará si el triángulo es Equilátero, Isósceles o Escaleno
 - **calculoArea():** Calculará el área del triángulo y lo devolverá. Utiliza la siguiente fórmula para ayudarte:

La fórmula de Herón es útil cuando conoces las longitudes de los tres lados:

$$A = \sqrt{s \cdot (s - a) \cdot (s - b) \cdot (s - c)}$$

donde:

- a, b, c son las longitudes de los lados.
- s es el semiperímetro, calculado como:

$$s = \frac{a + b + c}{2}$$

Encapsulación

Encapsulación

La encapsulación de datos es el pilar básico de la programación orientada a objetos, que consiste en proteger los datos de accesos o usos no controlados.

Los datos (atributos) que componen una clase pueden ser de dos tipos:

- **Públicos:** los datos son accesibles sin control.
- **Privados:** los datos son accesibles de forma controlada.

Encapsulación

Para poder encapsular los datos lo que se tiene que hacer es definirlos como privados y generar un método en la clase para poder acceder a ellos, de esta forma, únicamente son accesibles de forma directa los datos por la clase.

La encapsulación no sólo puede realizarse sobre los atributos de la clase, también es posible realizarla sobre los métodos, es decir, aquellos métodos que indiquemos que son privados no podrán ser utilizados por elementos externos al propio objeto.

Clase con atributos públicos

```
class PuntoPublico:  
    def __init__(self, x, y):  
        self.X = x  
        self.Y = y
```

Podríamos acceder a los valores de forma normal como hemos hecho hasta ahora:

```
publico = PuntoPublico(4,6)  
print("Valores punto publico:", publico.X,",",publico.Y)  
publico.X = 2  
print("Valores punto publico:", publico.X,",",publico.Y)
```

```
Valores punto publico: 4 , 6
```

```
Valores punto publico: 2 , 6
```

Clase con atributos privados

class PuntoPrivado:

def __init__ (self, x, y):

self.__X = x

self.__Y = y

def GetX(self):

return self.__X

def GetY(self):

return self.__Y

def SetX(self, x):

self.__X = x

def SetY(self, y):

self.__Y = y

Para definir un punto, lo hacemos igual:

privado = PuntoPrivado(7,3)

Sin embargo, para acceder a sus atributos, tenemos que usar los métodos GetX y GetY. Estos son llamados los **getters**.

print("Valores punto privado:", privado.GetX(),"",privado.GetY())

De igual manera, si queremos modificar los valores de sus atributos, usamos los métodos SetX y SetY. Estos son los llamados **setters**.

privado.SetX(9)

print("Valores punto privado:", privado.GetX(),"",privado.GetY())

Clase con métodos privados

De igual manera, podemos encapsular los métodos de una clase. La definición de métodos privados se realiza incluyendo los caracteres “__” delante del nombre del método. Vamos a definir dos métodos privados y uno público el cual utilizará los dos privados.

```
class OperarValores:
    def __init__(self, v1, v2):
        self.__V1 = v1
        self.__V2 = v2
    def __Sumar(self):
        return self.__V1 + self.__V2
    def __Restar(self):
        return self.__V1 - self.__V2
    def Operar(self):
        print("El resultado de la suma es: ",self.__Sumar())
        print("El resultado de la resta es: ",self.__Restar())
```

Clase con métodos privados

Lo probaríamos de la siguiente manera:

```
operarvalores = OperarValores(7,3)
operarvalores.Operar()
```

Siendo el resultado por pantalla:

```
El resultado de la suma es:  10
El resultado de la resta es:  4
>>>
```

Si intentamos ejecutar el siguiente código, obtenemos lo siguiente.

```
print("El resultado de la suma es :", operarvalores.__Sumar())
```

```
Traceback (most recent call last):
  File "/Users/alfre/Desktop/Python/7-1-3.py", line 15, in <module>
    print("El resultado de la suma es :",operarvalores.__Sumar())
AttributeError: 'OperarValores' object has no attribute '__Sumar'
>>>
```

Herencia

Herencia

La herencia consiste en la definición de una clase utilizando como base una clase ya existente. La nueva clase derivada tendrá todas las características de la clase base y ampliará el concepto de ésta, es decir, tendrá todos los atributos y métodos de la clase base. Por tanto, la herencia nos va a permitir reutilizar código fuente.

Cuando hablemos de herencia tienes que pensar que entre las dos clases existe una relación del tipo "es un".

Herencia

Lo primero que haremos será la utilización de una clase que será heradada por otra. La clase que será heradada será una clase que llamaremos Electrodomestico, que contendrá una serie de atributos y métodos que pueden ser heredados por otros electrodomésticos más concretos, como puede ser la clase Lavadora.

Primero definimos la clase Electrodomestico.

Herencia

```
class Electrodomestico:
```

```
    def __init__ (self):
```

```
        self.__Encendido = False
```

```
        self.__Tension = 0
```

```
    def Encender(self):
```

```
        self.__Encendido = True
```

```
    def Apagar(self):
```

```
        self.__Encendido = False
```

```
    def Encendido(self):
```

```
        return self.__Encendido
```

```
    def SetTension(self, tension):
```

```
        self.__Tension = tension
```

```
    def GetTension(self):
```

```
        return self.__Tension
```

Herencia

A continuación, definiremos la clase Lavadora. Para realizar la operación de herencia en Python únicamente hay que añadir entre paréntesis en la cabecera de la definición de clase la clase de la que queremos que herede.

Herencia

```
class Lavadora(Electrodomestico) :  
    def __init__ (self):  
        self.__RPM = 0  
        self.__Kilos = 0  
    def SetRPM(self, rpm):  
        self.__RPM = rpm  
    def SetKilos(self, kilos):  
        self.__Kilos = kilos  
    def MostrarLavadora(self):  
        print("Lavadora:")  
        print("RPM:",self.__RPM)  
        print("Kilos:",self.__Kilos)  
        print("Tension:",self.GetTension() )  
        if self.Encendido() :  
            print("¡Lavadora encendida!")
```


Herencia

A la hora de crear una clase derivada, es recomendable el uso de **super()**:

```
class Animal:
```

```
    def __init__(self, nombre):
```

```
        self.nombre = nombre
```

```
    def hacer_sonido(self):
```

```
        return "Sonido genérico"
```

```
class Perro(Animal):
```

```
    def __init__(self, nombre, raza):
```

```
        super().__init__(nombre) # Llama al constructor de la clase padre
```

```
        self.raza = raza
```

```
    def hacer_sonido(self):
```

```
        return "Guau"
```

__str__

__repr__

__eq__

__lt__

Imprimir objetos por pantalla: `__str__`

Cuando imprimimos un objeto por pantalla, nos sale de la siguiente forma:

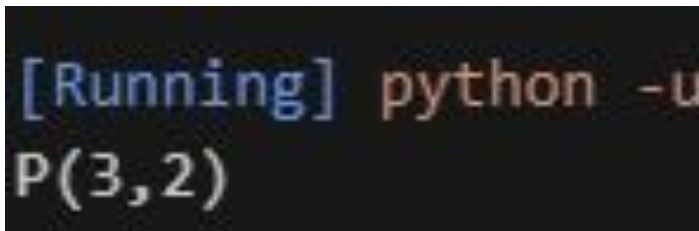
```
<__main__.Punto object at 0x0000016C2F4D5D90>
```

Para que nos salga de la manera que queramos, debemos de definir el método `__str__`:

```
class Punto:
    def __init__(self, x, y):
        self.X = x
        self.Y = y

    def __str__(self):
        return f"P({self.X},{self.Y})"
```

```
punto1 = Punto(3,2)
print(punto1)
```



A terminal window with a dark background. The first line shows the command `[Running] python -u` in blue and red text. The second line shows the output `P(3,2)` in white text.

Imprimir objetos por pantalla: `__str__`

Sin embargo, si por ejemplo hacemos una lista de puntos e intentamos imprimirla por pantalla, nos saldrá lo siguiente:

```
<__main__.Punto object at 0x000002AECF8D61B0>, <__main__.Punto object at 0x000002AECF8D6210>, <__main__.Punto object at 0x000002AECF8D61E0>, <__main__.Punto object at 0x000002AECF8D6300>
```

Esto ocurre porque Python usa el método `__repr__` dentro de cada objeto dentro de la lista, por lo que tendríamos que definirlo (ponemos lo mismo que en `__str__`):

```
def __repr__(self):  
    return f"P({self.X}, {self.Y})"
```

```
punto1 = Punto(3,5)
```

```
punto2 = Punto(3,7)
```

```
punto3 = Punto(2,9)
```

```
print([punto1, punto2, punto3])
```

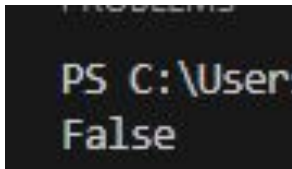
```
[P(3,5), P(3,7), P(2,9)]
```

Comprobar igualdad entre 2 objetos

Al ser los objetos de clases tipos de datos más complejos, no podemos compararlos directamente. Para ello, tenemos que definir el método `__eq__`, en el que definiremos cuándo dos objetos de una misma clase son iguales.

Si ahora creamos 2 puntos con las mismas coordenadas y los comparamos, tendremos el siguiente resultado:

```
punto1 = Punto(3,5)
punto2 = Punto(3,5)
print(punto1 == punto2)
```



```
PS C:\User:
False
```

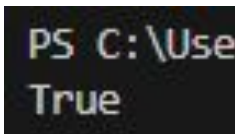
Comprobar igualdad entre 2 objetos

Por lo que, dentro de la clase Punto, definimos el método `__eq__` y vemos que ahora la comparación si se hace correctamente:

```
class Punto:
    def __init__(self, x, y):
        self.X = x
        self.Y = y

    def __eq__(self, otroPunto):
        return self.X == otroPunto.X and self.Y == otroPunto.Y
```

```
punto1 = Punto(3,5)
punto2 = Punto(3,5)
print(punto1 == punto2)
```



PS C:\Use
True

A screenshot of a terminal window with a black background. The text 'PS C:\Use' is on the first line and 'True' is on the second line, both in a light blue or cyan monospaced font.

Ordenar objetos

Otra cosa que podemos hacer es ordenar objetos de una clase, o establecer qué condición es la que indica que un objeto es mayor que otro.

Si intentamos ordenar una lista de puntos, obtenemos el siguiente error:

```
punto1 = Punto(3,7)
punto2 = Punto(3,5)
punto3 = Punto(8,3)
punto4 = Punto(2,9)
lista = [punto1, punto2, punto3, punto4]
lista.sort()
print(listaPuntos)
```

```
Traceback (most recent call last):
  File "c:\Users\alvar\Downloads\ej5\prueba.py", line 22, in <module>
    listaPuntos.sort()
TypeError: '<' not supported between instances of 'Punto' and 'Punto'
```

Ordenar objetos

Para solucionar esto, debemos definir el método `__lt__`. Significa (less than), por lo que estaremos definiendo que significa que un objeto sea menor que otro. Por ejemplo, si queremos que la coordenada X sea lo que indica que un objeto es menor que otro:

```
def __lt__(self, value):  
    return self.X < value.X
```

Ahora si ejecutamos el código de antes, nos saldrá la lista ordenada según la coordenada X.

```
[P(2,9), P(3,7), P(3,5), P(8,3)]
```

Si por ejemplo quisiéramos, en caso de empate en la coordenada X, se mirará la coordenada Y, podríamos hacerlo de la siguiente manera:

```
def __lt__(self, value):  
    if self.X == value.X:  
        return self.Y < value.Y  
    return self.X < value.X
```

```
[P(2,9), P(3,5), P(3,7), P(8,3)]
```