

		Guía Teórica		
CÓDIGO SI315	ASIGNATURA: Programación III	PRELACIÓN AS215	UC 5	Profesor: Guido Portillo

Programación Orientada a Objetos

Antes de nada, empecemos con una introducción básica a la Programación Orientada a Objetos POO o OOP en inglés. Se trata de un paradigma de programación introducido en los años 1970s, pero que no se hizo popular hasta años más tarde.

Este modo o paradigma de programación nos permite organizar el código de una manera que se asemeja bastante a como pensamos en la vida real, utilizando las famosas clases. Estas nos permiten agrupar un conjunto de variables y funciones que veremos a continuación.

Cosas de lo más cotidianas como un perro o un coche pueden ser representadas con clases. Estas clases tienen diferentes características, que en el caso del perro podrían ser la edad, el nombre o la raza. Llamaremos a estas características, atributos.

Por otro lado, las clases tienen un conjunto de funcionalidades o cosas que pueden hacer. En el caso del perro podría ser andar o ladrar. Llamaremos a estas funcionalidades métodos.

Por último, pueden existir diferentes tipos de perro. Podemos tener uno que se llama Toby o el del vecino que se llama Laika. Llamaremos a estos diferentes tipos de perro objetos. Es decir, el concepto abstracto de perro es la clase, pero Toby o cualquier otro perro particular será el objeto.

La programación orientada a objetos está basada en 6 principios o pilares básicos:

- Herencia
- Cohesión
- Abstracción
- Polimorfismo
- Acoplamiento
- Encapsulamiento

Motivación

Una vez explicada la programación orientada a objetos puede parecer bastante lógica, pero no es algo que haya existido siempre, y de hecho hay muchos lenguajes de programación que no la soportan.

En parte surgió debido a la creciente complejidad a la que los programadores se iban enfrentando conforme pasaban los años. En el mundo de la programación hay gran cantidad de aplicaciones que realizan tareas muy similares y es importante identificar los patrones que nos permiten no reinventar la rueda. La programación orientada a objetos intentaba resolver esto.

Uno de los primeros mecanismos que se crearon fueron las funciones, que permiten agrupar bloques de código que hacen una tarea específica bajo un nombre. Algo muy útil ya que permite también reusar esos módulos o funciones sin tener que copiar todo el código, tan solo la llamada.

Las funciones resultaron muy útiles, pero no eran capaces de satisfacer todas las necesidades de los programadores. Uno de los problemas de las funciones es que sólo realizan unas operaciones con unos datos de entrada para entregar una salida, pero no les importa demasiado conservar esos datos o mantener algún tipo de estado. Salvo que se devuelva un valor en la llamada a la función o se usen variables globales, todo lo que pasa dentro de la función queda olvidado, y esto en muchos casos es un problema.

Imaginemos que tenemos un juego con naves espaciales moviéndose por la pantalla. Cada nave tendrá una posición (x,y) y otros parámetros como el tipo de nave, su color o tamaño. Sin hacer uso de clases y POO, tendremos que tener una variable para cada dato que queremos almacenar: coordenadas, color, tamaño, tipo. El problema viene si tenemos 10 naves, ya que nos podríamos juntar con un número muy elevado de variables. Todo un desastre.

En el mundo de la programación existen tareas muy similares al ejemplo con las naves, y en respuesta a ello surgió la programación orientada a objetos. Una herramienta perfecta que permite resolver cierto tipo de problemas de una forma mucho más sencilla, con menos código y más organizado. Agrupa bajo una clase un conjunto de variables y funciones, que pueden ser reutilizadas con características particulares creando objetos.

Definiendo clases

Vista ya la parte teórica, vamos a ver como podemos hacer uso de la programación orientada a objetos en Python. Lo primero es crear una clase, para ello usaremos el ejemplo de perro.

```
# Creando una clase vacía
class Perro:
    pass
```

Se trata de una clase vacía y sin mucha utilidad práctica, pero es la mínima clase que podemos crear. Nótese el uso del `pass` que no hace realmente nada, pero daría un error si después de los `:` no tenemos contenido.

Ahora que tenemos la clase, podemos crear un objeto de la misma. Podemos hacerlo como si de una variable normal se tratase. Nombre de la variable igual a la clase con `()`. Dentro de los paréntesis irían los parámetros de entrada si los hubiera.

```
# Creamos un objeto de la clase perro
mi_perro = Perro()
```

Definiendo atributos

A continuación vamos a añadir algunos atributos a nuestra clase. Antes de nada es importante distinguir que existen dos tipos de atributos:

- Atributos de instancia: Pertenecen a la instancia de la clase o al objeto. Son atributos particulares de cada instancia, en nuestro caso de cada perro.
- Atributos de clase: Se trata de atributos que pertenecen a la clase, por lo tanto serán comunes para todos los objetos.

Empecemos creando un par de atributos de instancia para nuestro perro, el `nombre` y la `raza`. Para ello creamos un método `__init__` que será llamado automáticamente cuando creamos un objeto. Se trata del constructor.

```
class Perro:
    # El método __init__ es llamado al crear el objeto
    def __init__(self, nombre, raza):
        print(f'Creando perro {nombre}, {raza}')

        # Atributos de instancia
        self.nombre = nombre
        self.raza = raza
```

Ahora que hemos definido el método `init` con dos parámetros de entrada, podemos crear el objeto pasando el valor de los atributos. Usando `type()` podemos ver como efectivamente el objeto es de la clase `Perro`.

```
mi_perro = Perro("Toby", "Bulldog")
print(type(mi_perro))
# Creando perro Toby, Bulldog
# <class '__main__.Perro'>
```

Seguramente te hayas fijado en el `self` que se pasa como parámetro de entrada del método. Es una variable que representa la instancia de la clase, y deberá estar siempre ahí.

El uso de `__init__` y el doble `__` no es una coincidencia. Cuando veas un método con esa forma, significa que está reservado para un uso especial del lenguaje. En este caso sería lo que se conoce como constructor. Hay gente que llama a estos métodos mágicos.

Por último, podemos acceder a los atributos usando el objeto y `.`. Por lo tanto.

```
print(mi_perro.nombre) # Toby  
print(mi_perro.raza)  # Bulldog
```

Hasta ahora hemos definido atributos de instancia, ya que son atributos que pertenecen a cada perro concreto. Ahora vamos a definir un atributo de clase, que será común para todos los perros. Por ejemplo, la especie de los perros es algo común para todos los objetos Perro.

```
class Perro:  
    # Atributo de clase  
    especie = 'mamífero'  
  
    # El método __init__ es llamado al crear el objeto  
    def __init__(self, nombre, raza):  
        print(f"Creando perro {nombre}, {raza}")  
  
    # Atributos de instancia  
    self.nombre = nombre  
    self.raza = raza
```

Dado que es un atributo de clase, no es necesario crear un objeto para acceder al atributo. Podemos hacer lo siguiente.

```
print(Perro.especie)  
# mamífero
```

Se puede acceder también al atributo de clase desde el objeto.

```
mi_perro = Perro("Toby", "Bulldog")  
mi_perro.especie  
# 'mamífero'
```

De esta manera, todos los objetos que se creen de la clase perro compartirán ese atributo de clase, ya que pertenecen a la misma.

En realidad cuando usamos `__init__` anteriormente ya estábamos definiendo un método, solo que uno especial. A continuación vamos a ver como definir métodos que le den alguna funcionalidad interesante a nuestra clase, siguiendo con el ejemplo de perro.

Vamos a codificar dos métodos, ladrar y caminar. El primero no recibirá ningún parámetro y el segundo recibirá el número de pasos que queremos andar. Como hemos indicado anteriormente `self` hace referencia a la instancia de la clase. Se puede definir un método con `def` y el nombre, y entre `()` los parámetros de entrada que recibe, donde siempre tendrá que estar `self` el primero.

```
class Perro:
    # Atributo de clase
    especie = 'mamífero'

    # El método __init__ es llamado al crear el objeto
    def __init__(self, nombre, raza):
        print(f"Creando perro {nombre}, {raza}")

        # Atributos de instancia
        self.nombre = nombre
        self.raza = raza

    def ladra(self):
        print("Guau")

    def camina(self, pasos):
        print(f"Caminando {pasos} pasos")
```

Por lo tanto si creamos un objeto `mi_perro`, podremos hacer uso de sus métodos llamándolos con `.` y el nombre del método. Como si de una función se tratase, pueden recibir y devolver argumentos.

```
mi_perro = Perro("Toby", "Bulldog")
mi_perro.ladra()
mi_perro.camina(10)

# Creando perro Toby, Bulldog
# Guau
# Caminando 10 pasos
```

Si te ha parecido útil este post, tal vez te interese:

- Uso del decorador `property`
- Métodos estáticos y de clase

Métodos en Python: instancia, clase y estáticos

En otros posts hemos visto como se pueden crear métodos con `def` dentro de una clase, pudiendo recibir parámetros como entrada y modificar el estado (como los atributos) de la instancia. Pues bien, haciendo uso de los decoradores, es posible crear diferentes tipos de métodos:

- Lo métodos de instancia “normales” que ya hemos visto como `metodo()`
- Métodos de clase usando el decorador `@classmethod`
- Y métodos estáticos usando el decorador `@staticmethod`

En la siguiente clase tenemos un ejemplo donde definimos los tres tipos de métodos.

```
class Clase:
    def metodo(self):
        return 'Método normal', self

    @classmethod
    def metododeclase(cls):
        return 'Método de clase', cls

    @staticmethod
    def metodoestatico():
        return "Método estático"
```

Veamos su comportamiento en detalle uno por uno.

Métodos de instancia

Los métodos de instancia son los métodos normales, de toda la vida, que hemos visto anteriormente. Reciben como parámetro de entrada `self` que hace referencia a la instancia que llama al método. También pueden recibir otros argumentos como entrada.

Para saber más: El uso de "self" es totalmente arbitrario. Se trata de una convención acordada por los usuarios de Python, usada para referirse a la instancia que llama al método, pero podría ser cualquier otro nombre. Lo mismo ocurre con "cls", que veremos a continuación.

```
class Clase:
    def metodo(self, arg1, arg2):
        return 'Método normal', self
```

Y como ya sabemos, una vez creado un objeto pueden ser llamados.

```
mi_clase = Clase()
mi_clase.metodo("a", "b")
# ('Método normal', <__main__.Clase at 0x10b9daa90>)
```

En vista a esto, los métodos de instancia:

- Pueden acceder y modificar los atributos del objeto.
- Pueden acceder a otros métodos.
- Dado que desde el objeto `self` se puede acceder a la clase con ``self.class``, también pueden modificar el estado de la clase

Métodos de clase (classmethod)

A diferencia de los métodos de instancia, los métodos de clase reciben como argumento `cls`, que hace referencia a la clase. Por lo tanto, pueden acceder a la clase pero no a la instancia.

```
class Clase:
    @classmethod
    def metododeclase(cls):
        return 'Método de clase', cls
```

Se pueden llamar sobre la clase.

```
Clase.metododeclase()
# ('Método de clase', __main__.Clase)
```

Pero también se pueden llamar sobre el objeto.

```
mi_clase.metododeclase()
# ('Método de clase', __main__.Clase)
```

Por lo tanto, los métodos de clase:

- No pueden acceder a los atributos de la instancia.
- Pero si pueden modificar los atributos de la clase.
-

Métodos estáticos (staticmethod)

Por último, los métodos estáticos se pueden definir con el decorador `@staticmethod` y no aceptan como parámetro ni la instancia ni la clase. Es por ello por lo que no pueden modificar el estado ni de la clase ni de la instancia. Pero por supuesto pueden aceptar parámetros de entrada.

```
class Clase:
    @staticmethod
    def metodoestatico():
        return "Método estático"
mi_clase = Clase()
Clase.metodoestatico()
mi_clase.metodoestatico()

# 'Método estático'
# 'Método estático'
```

Por lo tanto el uso de los métodos estáticos pueden resultar útil para indicar que un método no modificará el estado de la instancia ni de la clase. Es cierto que se podría hacer lo mismo con un método de instancia por ejemplo, pero a veces resulta importante indicar de alguna manera estas peculiaridades, evitando así futuros problemas y malentendidos.

En otras palabras, los métodos estáticos se podrían ver como funciones normales, con la salvedad de que van ligadas a una clase concreta.

Herencia en Python

Para entender la herencia, es fundamental entender la programación orientada a objetos, por lo que te recomendamos empezar por ahí antes.

La herencia es un proceso mediante el cual se puede crear una clase hija que hereda de una clase padre, compartiendo sus métodos y atributos. Además de ello, una clase hija puede sobrescribir los métodos o atributos, o incluso definir unos nuevos.

Se puede crear una clase hija con tan solo pasar como parámetro la clase de la que queremos heredar. En el siguiente ejemplo vemos como se puede usar la herencia en Python, con la clase `Perro` que hereda de `Animal`. Así de fácil.

```
# Definimos una clase padre
```

```
class Animal:
    pass

# Creamos una clase hija que hereda de la padre
class Perro(Animal):
    pass
```

De hecho podemos ver como efectivamente la clase `Perro` es la hija de `Animal` usando `__bases__`

```
print(Perro.__bases__)
# (<class '__main__.Animal'>,)
```

De manera similar podemos ver que clases descienden de una en concreto con `__subclasses__`.

```
print(Animal.__subclasses__())
# [<class '__main__.Perro'>]
```

¿Y para que queremos la herencia? Dado que una clase hija hereda los atributos y métodos de la padre, nos puede ser muy útil cuando tengamos clases que se parecen entre sí pero tienen ciertas particularidades. En este caso en vez de definir un montón de clases para cada animal, podemos tomar los elementos comunes y crear una clase `Animal` de la que hereden el resto, respetando por tanto la filosofía DRY. Realizar estas abstracciones y buscar el denominador común para definir una clase de la que hereden las demás, es una tarea de lo más compleja en el mundo de la programación.

Para saber más: El principio DRY (Don't Repeat Yourself) es muy aplicado en el mundo de la programación y consiste en no repetir código de manera innecesaria. Cuanto más código duplicado exista, más difícil será de modificar y más fácil será crear inconsistencias. Las clases y la herencia a no repetir código.

Extendiendo y modificando métodos

Continuemos con nuestro ejemplo de perros y animales. Vamos a definir una clase padre `Animal` que tendrá todos los atributos y métodos genéricos que los animales pueden tener. Esta tarea de buscar el denominador común es muy importante en programación. Veamos los atributos:

- Tenemos la especie ya que todos los animales pertenecen a una.
- Y la edad, ya que todo ser vivo nace, crece, se reproduce y muere.

Y los métodos o funcionalidades:

- Tendremos el método hablar, que cada animal implementará de una forma. Los perros ladran, las abejas zumban y los caballos relinchan.
- Un método moverse. Unos animales lo harán caminando, otros volando.
- Y por último un método describeme que será común.

Definimos la clase padre, con una serie de atributos comunes para todos los animales como hemos indicado.

```
class Animal:
    def __init__(self, especie, edad):
        self.especie = especie
        self.edad = edad

    # Método genérico pero con implementación particular
    def hablar(self):
        # Método vacío
        pass

    # Método genérico pero con implementación particular
    def moverse(self):
        # Método vacío
        pass

    # Método genérico con la misma implementación
    def describeme(self):
        print("Soy un Animal del tipo", type(self).__name__)
```

Tenemos ya por lo tanto una clase genérica `Animal`, que generaliza las características y funcionalidades que todo animal puede tener. Ahora creamos una clase `Perro` que hereda del `Animal`. Como primer ejemplo vamos a crear una clase vacía, para ver como los métodos y atributos son heredados por defecto.

```
# Perro hereda de Animal
class Perro(Animal):
    pass

mi_perro = Perro('mamífero', 10)
mi_perro.describeme()
# Soy un Animal del tipo Perro
```

Con tan solo un par de líneas de código, hemos creado una clase nueva que tiene todo el contenido que la clase padre tiene, pero aquí viene lo que es de verdad interesante. Vamos a crear varios animales concretos y sobrescribir algunos de los métodos que habían sido definidos en la clase `Animal`, como el `hablar` o el `moverse`, ya que cada animal se comporta de una manera distinta.

Podemos incluso crear nuevos métodos que se añadirán a los ya heredados, como en el caso de la Abeja con `picar()`.

```
class Perro(Animal):
    def hablar(self):
        print("Guau!")
    def moverse(self):
        print("Caminando con 4 patas")

class Vaca(Animal):
    def hablar(self):
        print("Muuu!")
    def moverse(self):
        print("Caminando con 4 patas")

class Abeja(Animal):
    def hablar(self):
        print("Bzzzz!")
    def moverse(self):
        print("Volando")

    # Nuevo método
    def picar(self):
        print("Picar!")
```

Por lo tanto ya podemos crear nuestros objetos de esos animales y hacer uso de sus métodos que podrían clasificarse en tres:

- Heredados directamente de la clase padre: `describeme()`
- Heredados de la clase padre pero modificados: `hablar()` y `moverse()`
- Creados en la clase hija por lo tanto no existentes en la clase padre: `picar()`

```
mi_perro = Perro('mamífero', 10)
mi_vaca = Vaca('mamífero', 23)
mi_abeja = Abeja('insecto', 1)
```

```
mi_perro.hablar()
mi_vaca.hablar()
# Guau!
# Muuu!
```

```
mi_vaca.describeme()
mi_abeja.describeme()
# Soy un Animal del tipo Vaca
```

```
# Soy un Animal del tipo Abeja
```

```
mi_abeja.picar()
```

```
# Picar!
```

Uso de super()

En pocas palabras, la función `super()` nos permite acceder a los métodos de la clase padre desde una de sus hijas. Volvamos al ejemplo de `Animal` y `Perro`.

```
class Animal:
    def __init__(self, especie, edad):
        self.especie = especie
        self.edad = edad
    def hablar(self):
        pass

    def moverse(self):
        pass

    def describeme(self):
        print("Soy un Animal del tipo", type(self).__name__)
```

Tal vez queramos que nuestro `Perro` tenga un parámetro extra en el constructor, como podría ser el `dueño`. Para realizar esto tenemos dos alternativas:

- Podemos crear un nuevo `__init__` y guardar todas las variables una a una.
- O podemos usar `super()` para llamar al `__init__` de la clase padre que ya aceptaba la `especie` y `edad`, y sólo asignar la variable nueva manualmente.

```
class Perro(Animal):
    def __init__(self, especie, edad, dueño):
        # Alternativa 1
        # self.especie = especie
        # self.edad = edad
        # self.dueño = dueño

        # Alternativa 2
        super().__init__(especie, edad)
        self.dueño = dueño
```

```
mi_perro = Perro('mamífero', 7, 'Luis')
mi_perro.especie
mi_perro.edad
mi_perro.dueño
```

Herencia múltiple

En Python es posible realizar herencia múltiple. En otros posts hemos visto como se podía crear una clase padre que heredaba de una clase hija, pudiendo hacer uso de sus métodos y atributos. La herencia múltiple es similar, pero una clase hereda de varias clases padre en vez de una sola.

Veamos un ejemplo. Por un lado tenemos dos clases `Clase1` y `Clase2`, y por otro tenemos la `Clase3` que hereda de las dos anteriores. Por lo tanto, heredará todos los métodos y atributos de ambas.

```
class Clase1:
    pass
class Clase2:
    pass
class Clase3(Clase1, Clase2):
    pass
```

Es posible también que una clase herede de otra clase y a su vez otra clase herede de la anterior.

```
class Clase1:
    pass
class Clase2(Clase1):
    pass
class Clase3(Clase2):
    pass
```

Llegados a este punto nos podemos plantear lo siguiente. Vale, como sabemos de otros posts las clases hijas heredan los métodos de las clases padre, pero también pueden reimplementarlos de manera distinta. Entonces, si llamo a un método que todas las clases tienen en común ¿a cuál se llama?. Pues bien, existe una forma de saberlo.

La forma de saber a que método se llama es consultar el MRO o *Method Order Resolution*. Esta función nos devuelve una tupla con el orden de búsqueda

de los métodos. Como era de esperar se empieza en la propia clase y se va subiendo hasta la clase padre, de izquierda a derecha.

```
class Clase1:
    pass
class Clase2:
    pass
class Clase3(Clase1, Clase2):
    pass

print(Clase3.__mro__)
# (<class '__main__.Clase3'>, <class '__main__.Clase1'>, <class
'__main__.Clase2'>, <class 'object'>)
```

Una curiosidad es que al final del todo vemos la clase `object`. Aunque pueda parecer raro, es correcto ya que en realidad todas las clases en Python heredan de una clase genérica `object`, aunque no lo especifiquemos explícitamente.

Y como último ejemplo,...el cielo es el límite. Podemos tener una clase heredando de otras tres. Fíjate en que el MRO depende del orden en el que las clases son pasadas: 1, 3, 2.

```
class Clase1:
    pass
class Clase2:
    pass
class Clase3:
    pass
class Clase4(Clase1, Clase3, Clase2):
    pass
print(Clase4.__mro__)
# (<class '__main__.Clase4'>, <class '__main__.Clase1'>, <class
'__main__.Clase3'>, <class '__main__.Clase2'>, <class 'object'>)
```

Junto con la herencia, la cohesión, abstracción, polimorfismo, acoplamiento y encapsulamiento son otros de los conceptos claves para entender la programación orientada a objetos.