Programación Orientada a Objetos

PROTECO

POO

La programación orientada a objetos (POO) es otro paradigma de programación. En este, el programa se va a estructurar de tal forma que las acciones y las propiedades necesarias para dar solución al problema estén vinculadas con un **objeto** particular.

Objetos

Un objeto puede representar muchas cosas. Por ejemplo, un carro. Los carros **tienen características**, estos tienen un color, tienen una marca, tienen un precio, etc. De igual forma, los carros **pueden realizar acciones**, los carros pueden avanzar, los carros pueden acelerar, los carros pueden hacer sonar su bocina, etc.

Otro ejemplo, una persona. Una persona tiene características como lo es su nombre, su edad, su dirección,etc. Y puede realizar acciones como caminar, hablar, respirar,etc.

Objetos

En general, con los objetos modelamos cosas de la vida real, estos objetos van a contar con características (atributos) y van a poder realizar acciones (métodos). Cada objeto va a contar con sus propias características y va a poder realizar acciones de una manera particular, por ejemplo, dos personas pueden tener distintas edades y pueden correr de distintas formas.

Python y POO

En Python todas las cosas son objetos, podemos tomar como ejemplo una lista. Las cuales, al igual que los objetos descritos anteriormente, tienen características y pueden realizar acciones. Como característica, las listas tienen una longitud. Como acción, las listas pueden agregar un elemento al final por medio del método append.

```
>>> lista1 = [2,3,4,5]
>>> lista1.append(6)
>>> lista1
[2, 3, 4, 5, 6]
```

Python y POO

Si nosotros creamos otra lista.

```
>>> lista2 = [1,2,1,2,1,2,1,2]
```

Podemos ver que las dos listas son del mismo tipo de objeto (misma clase), ambas son listas, sin embargo, al igual que dos personas no van a ser iguales, cada persona tiene su propia edad y su propio nombre, estas listas tampoco lo son, cada una tiene su propia longitud y sus propios elementos.

Clases

Como se ha dicho, las listas, los strings y los demás tipos de datos tienen su propia clase. Las listas pertenecen a la clase list.

```
>>> type([1,2,3,4])
<class 'list'>
```

Las clases son la idea de cómo van a funcionar los objetos, la clase list es la idea de que las listas van a poder agregar elementos al final con append, que van a poder eliminar elementos con pop, que van a tener una longitud determinada, etc.

Clases

Las clases son la idea o la estructura que va a tener un objeto, la clase list nos va a decir que van a poder hacer las listas y qué características van a poder tener, sin embargo, solo es una idea, no es una lista como tal.

Clases

Python nos proporciona con tipos de datos (clases) como entero, flotante, listas, tuplas, diccionarios, etc. Con la clase int podemos representar una calificación, una edad, el costo de algo,etc. Con la clase string podríamos representar el nombre de alguien, una frase, etc. Sin embargo, imaginemos que queremos representar algo más complicado, como los es un coche. Para esto, nosotros podemos crear nuestra propia clase coche.

Clases

Para definir una clase en Python, se utiliza la siguiente sintaxis.

Se coloca la palabra class seguido del nombre de la clase, dos puntos y en otro bloque los métodos y atributos.

```
class Coche:
    def avanzar(self):
        print('Rummmm')

    def sonarBocina(self):
        print('PIIIP!')
```

Acabamos de crear una clase Coche, que va a poder realizar las acciones avanzar y sonar Bocina, ambas van a realizar una impresión. Podemos ver que las acciones se definen como las funciones, pero tienen como **parámetro inicial self**.

Clases

Una vez definida la clase, al igual que podemos crear listas o diccionarios, ahora podemos crear coches. Más exactamente, podemos crear objetos que pertenezcan a la clase coche. A esta creación de objetos se le denomina instanciación.

```
>>> class Coche:
...     def avanzar(self):
...             print('Rummm')
...
...     def sonarBocina(self):
...             print('PIIPP!!')
...
>>> coche1 = Coche()
```

Se tiene que poner el nombre de la clase seguido de paréntesis, y eso lo asignamos a una variable.

Métodos

Los métodos de un objeto se pueden mandar a llamar, colocando la variable que contiene al objeto, un punto, el nombre del método y dos paréntesis.

```
>>> coche1 = Coche()
>>> coche2 = Coche()
>>> coche1.avanzar()
Rumm
>>> coche2.sonarBocina()
PIIPP!
```

Aquí creamos dos coches, el primero realizó la acción de avanzar y el segundo la de sonarBocina.

self

La palabra self va a denotar la instancia, o el objeto particular. Cuándo coche1 realiza el método avanzar, self es coche1, cuándo coche2 realiza el método sonarBocina, self es coche2. Es necesario que cuando definamos un método, su primer parámetro sea self.

```
>>> coche1 = Coche()
>>> coche2 = Coche()
>>> coche1.avanzar()
Rumm
>>> coche2.sonarBocina()
PIIPP!
```

```
class Coche:
    def avanzar(self):
        print('Rummmm')

    def sonarBocina(self):
        print('PIIIP!')
```

Atributos

Las clases y los objetos pueden tener características, como lo es la edad o el nombre de una persona o el color y la raza de un perro. A estos les llamamos atributos. Para definirlos en nuestras clases, se declaran como una variable normal, por ejemplo, con el código siguiente hacemos que todos los coches tengan 4 llantas y pesen 800 kg.

Para ver el atributo de una clase u objeto, se coloca el nombre de la clase u objeto, punto y el nombre del atributo. Los atributos creados sn atributos de la clase.

```
>>> class Coche:
...     llantas = 4
...     peso = 800
...     def sonarBocina(self):
...             print('PIIIP!!')
...
>>> Coche.llantas
4
>>> Coche.peso
800
```

Constructor

Para poder definir atributos específicos de cada objeto, primero tenemos que introducir el concepto de un constructor. Este es un método que se manda a llamar cuando creamos un objeto. Al igual que los métodos normales, su primer parámetro debe ser self, seguido de los parámetros que queremos convertir en atributos específicos del objeto. La principal diferencia con respecto a los otros métodos es que se debe nombrar de una forma particular, que es __init__. (dos guiónes bajos, init, dos guiónes bajos).

Constructor

Este constructor va recibir EDAD y RAZA, y los valores de estos los va a asignar a los atributos específicos de cada objeto, es por esto que necesitamos hacer uso de self, que simboliza el objeto particular.

```
>>> class Perro:
...     def __init__(self,EDAD,RAZA):
...             self.edad = EDAD
...             self.raza = RAZA
```

Constructor

Cuando creamos los objetos, se manda a llamar el constructor y podemos pasar argumentos a este, en este caso, edad y raza. El argumento para self no es necesario.

De esta manera, podemos hacer que cada perro tenga una edad y una raza en particular, siendo que perro1 es de raza Labrador y perro2 es de raza Collie.

```
>>> class Perro:
...     def __init__(self,EDAD,RAZA):
...             self.edad = EDAD
...             self.raza = RAZA
...
...     def ladrar(self):
...             print('WOOF!')
...
>>> perro1 = Perro(4,'Labrador')
>>> perro2 = Perro(2,'Collie')
>>> perro1.edad
4
>>> perro2.raza
'Collie'
>>> perro2.ladrar()
WOOF!
```

Abstracción

En la clase Coche que definimos anteriormente, únicamente definimos el método avanzar y sonar Bocina, sabemos que los coches pueden realizar muchas más acciones, como frenar o dar vuelta, sin embargo, especificar todas los métodos que podría realizar sería imposible, es por esto que solo ponemos los que nos interesa, a esta acción se le conoce como **abstracción** y es uno de los 4 pilares de la programación orientada a objetos.

```
>>> class Coche:
...     def avanzar(self):
...             print('Rummm')
...
...     def sonarBocina(self):
...             print('PIIPP!!')
...
>>> coche1 = Coche()
```

Ejemplo

```
>>> class Perro:
...     def __init__(self, Nombre, Alimento):
...             self.nombre = Nombre
...             self.alimento = Alimento
...             self.hambre = True
...
...     def comer(self):
...             if self.hambre:
...                     print(self.nombre,'está comiendo',self.alimento)
...                     self.hambre = False
...             else:
...                     print(self.nombre,'ya no tiene hambre')
...
>>> perro1 = Perro('Luna','Dog Chow')
>>> perro2 = Perro('Maya','Pollo')
>>> perro1.comer()
Luna está comiendo Dog Chow
>>> perro1.comer()
Luna ya no tiene hambre
>>> perro2.nombre
'Maya'
>>> perro2.comer()
Maya está comiendo Pollo
```

Métodos mágicos

Si nosotros creamos dos objetos lista como los siguientes.

```
>>> lista1 = [1,4,3]
>>> lista2 = [6,7,8]
```

Podemos realizar una suma de estos y nos va a regresar otro objeto lista que tenga una combinación de los elementos de las otras dos.

```
>>> lista1 + lista2
[1, 4, 3, 6, 7, 8]
```

Las listas tienen preestablecido que cuando se suman, van a retornar una nueva con la combinación de elementos. ¿Qué pasaría si nosotros tratamos de sumar dos de nuestras clases?

Métodos mágicos

Los métodos mágicos son métodos que se van a realizar cuando nosotros realizamos operaciones con objetos, por ejemplo, cuando sumamos dos objetos, cuando multiplicamos dos objetos o cuando comparemos dos objetos. Al igual que el constructor estos deben tener un nombre particular dependiendo del operador para el que se quiere programar.

A continuación vamos a definir una clase perro. Vamos a especificar que cuando dos objetos perro se sumen, se va a retornar la suma de sus edades.

Métodos mágicos

___

Para definir la suma, el método mágico se debe llamar __add__

```
>>> class Perro:
...     def __init__(self,NOMBRE,EDAD):
...             self.nombre = NOMBRE
...             self.edad = EDAD
...
...     def __add__(self,perro2):
...             return self.edad + perro2.edad
```

Aquí el método mágico __add__ está especificando que cuando dos perros se sumen se va a retornar la suma de sus edades.

Métodos mágicos

```
>>> class Perro:
...     def __init__(self,NOMBRE,EDAD):
...             self.nombre = NOMBRE
...             self.edad = EDAD
...
...     def __add__(self,perro2):
...             return self.edad + perro2.edad
...
>>> perro1 = Perro('Maya',3)
>>> perro2 = Perro('Luna',4)
>>> perro1 + perro2
7
```

Métodos mágicos

Existen muchas formas de definir comportamiento para diferentes operadores, como para la resta, la multiplicación, la comparación, etc. El nombre de los métodos mágicos son los siguientes:

Métodos mágicos

Operator	Método mágico
+	`object.__add__(self, other)`
-	`object.__sub__(self, other)`
*	`object.__mul__(self, other)`
//	`object.__floordiv__(self, other)`
/	`object.__truediv__(self, other)`
%	`object.__mod__(self, other)`
**	`object.__pow__(self, other[, modulo])`
<<	`object.__lshift__(self, other)`
>>	`object.__rshift__(self, other)`
&	`object.__and__(self, other)`
^	`object.__xor__(self, other)`
	`object.__or__(self, other)`

Métodos mágicos

Operator	Método mágico
-	`object.__neg__(self)`
+	`object.__pos__(self)`
abs()	`object.__abs__(self)`
~	`object.__invert__(self)`
complex()	`object.__complex__(self)`
int()	`object.__int__(self)`
long()	`object.__long__(self)`
float()	`object.__float__(self)`
oct()	`object.__oct__(self)`
hex()	`object.__hex__(self`

Métodos mágicos

Operadores de comparación

Operator	Método mágico
<	`object.__lt__(self, other)`
<=	`object.__le__(self, other)`
==	`object.__eq__(self, other)`
!=	`object.__ne__(self, other)`
>=	`object.__ge__(self, other)`
>	`object.__gt__(self, other)`

Herencia

Imaginemos que tenemos que definir dos clases, la clase animal y la clase perro, los animales van a poder respirar y alimentarse, mientras que los perros van a poder ladrar. Ahora, puesto que los perros también son animales, estos también van a poder respirar y alimentarse.

Se dice que los perros heredan de la clase animal las acciones de respirar y alimentarse.

Herencia

La **herencia** es el hecho de que una clase puede adquirir todos los métodos y atributos establecidos en otra. Para que una clase herede de otra, al definir la nueva clase, después de su nombre se deben colocar paréntesis y dentro de estos el nombre de la clase de la que hereda.

```python
class Animal:

    def respirar(self):
        print("Estoy respirando")

    def alimentarse(self):
        print("Me estoy alimentando")

class Perro(Animal):

    def ladrar(self):
        print("Woof!")
```

Herencia

```
>>> class Animal:
...     def respirar(self):
...             print('Estoy respirando')
...
...     def alimentarse(self):
...             print('Me estoy alimentando')
...
>>> class Perro(Animal):
...     def ladrar(self):
...             print('Woof!')
...
>>> perro1 = Perro()
>>> perro1.ladrar()
Woof!
>>> perro1.respirar()
Estoy respirando
>>> perro1.alimentarse()
Me estoy alimentando
```

Herencia múltiple

Un objeto puede heredar de múltiples clases. A esto se le conoce como multi herencia. En este caso, la clase anfibio hereda de acuático y de terrestre.

```python
class Terrestre:

    def respirar(self):
        print('Respiro fuera del agua')

    def caminar(self):
        print('Estoy caminando')

class Acuatico:

    def respirar(self):
        print('Respiro bajo el agua')

    def nadar(self):
        print('Estoy nadando')

class Anfibio(Acuatico,Terrestre):

    def saludar(self):
        print('Hola, soy un anfibio')
```

Polimorfismo

El **polimorfismo** al igual que la herencia y la abstracción es un pilar de la programación orientada a objetos. Esta se se basa en el hecho de que una misma acción se va a realizar de diferentes formas dependiendo del objeto que la realice.

Polimorfismo

En este caso, ambos objetos realizan el método hablar, sin embargo, dependiendo del objeto, es el comportamiento del método.

```
>>> class Perro:
...     def hablar(self):
...             print('Woof!')
...
>>> class Gato:
...     def hablar(self):
...             print('Miaw!')
...
>>> objeto1 = Perro()
>>> objeto2 = Gato()
>>> objeto1.hablar()
Woof!
>>> objeto2.hablar()
Miaw!
```