Introducción, Clase 1

Teoría de la Programación Orientada a Objetos

**Objetivo del 1º tutorial de Curso de Programación Orientada a Objetos en Python**

Comprender los conceptos que conforman la programación orientada a objetos y todas sus ventajas.

Para empezar debes comprender que hay dos tipos de formas de programar, la programación orientada a procedimiento (**POP**) y la programación orientada a objetos (**POO**). A grandes rasgos podríamos decir que los lenguajes de **POP** son los lenguajes antiguos y los lenguajes de **POO** son los lenguajes modernos, por lo tanto conviene diferenciar las características, ventajas y desventajas que tienen uno del otro.

**Programación orientada a procedimientos (POP)**

Algunos ejemplos de lenguajes de programación orientada a procedimientos son: Fortran, Cobol, Basic, etc. Estos lenguajes de programación ofrecían varias desventajas que promovieron a la creación de la programación orientada a objetos las cuales son:

* Largas líneas de código en aplicaciones largas y complejas.
* En aplicaciones complejas el código resultaba difícil de descifrar.
* Poca reutilización de código.
* Si existe un fallo en alguna línea de código, es muy probable que el programa caiga.
* Aparición frecuente de código espagueti.
* Difícil de depurar por otros programadores en caso de necesidad o error.

Ante todas estas dificultades a medida que pasa el tiempo surge la necesidad de crear aplicaciones más complejas y se decidió dar un giro de 180 grados e inventar una forma de programar totalmente diferente con el objetivo de que todas esas desventajas desaparezcan o por lo menos no sean tan pronunciadas y se inventó lo que se conoce como la programación orientada a objetos.

**Programación orientada a objetos (POO)**

Esta forma de programar consiste en trasladar la naturaleza de los objetos de la vida real a código de programación, esto recoge todas las funcionalidades y características que tiene un objeto tales como un estado, un comportamiento y sus atributos para así poder crear programas más útiles.

Para dar un ejemplo de objeto usaremos un coche y te preguntaremos lo siguiente:

* ¿Cuál es el estado de un coche? Un coche puede estar parado, circulando, aparcado, etc.
* ¿Qué atributos tiene un coche? Un coche tiene un color, un peso, un tamaño, etc.
* ¿Qué comportamiento tiene un coche? Un coche puede arrancar, frenar, acelerar, girar, etc.

Sabiendo todo esto gracias a la **POO** puedes trasladarlo al código de un programa y algunos ejemplos de leguajes de **POO** son: C++, Java, Visual.NET, Python, etc.

La **POO** nos ofrece las siguientes ventajas:

* Programas divididos en “trozos”, “partes”, “módulos” o “clases” permitiendo la modularización del código.
* Muy reutilizable implementando el concepto de herencia.
* Si existe un fallo en alguna línea de código el programa aun así podrá ejecutarse.
* Proporciona la encapsulación un método que permite la implementar protección al código para no ser usado fuera de la función donde está escrito.

**Conceptos de la POO**

La **POO** ofrece ciertos términos que a simple vista no son fáciles de entender, en este curso veras los más importantes para que la puedas entenderla lo mejor posible.

**Clase**

Una clase lo podríamos definir como un módulo donde se redactan las características comunes de un grupo de objetos. Ejemplo: Un objeto es un vehículo y sus clase son los tipos de vehículos tales como aviones, coche, trenes, motos, etc. Cada uno posee características únicas de su clase.

**Objeto**

Un objeto es un ejemplar perteneciente a una clase. Ejemplo: Si tenemos 2 vehículos un coche y una moto, ambos son objetos y pertenecen a dos clase diferentes, la clase moto y la clase coche.

**Atributo**

Son las características que tiene una clase.

**Comportamiento**

Son las acciones u operaciones que un objeto puede realizar.

**Modularización**

Normalmente cuando creas una aplicación compleja en un leguaje de **POO** como python es normal que esas aplicaciones estén compuestas de varias clases que trabajan de manera totalmente independiente una de otra y si una de las clases por cualquier razón deja de funcionar el programa no caerá eso es modularización.

**Encapsulación**

Es una función que impide que el funcionamiento de una clase no pueda ser usado o entendido por otra clase, es decir, por ejemplo: Un equipo de sonido no puede usar o tener las características de una nevera, por lo tanto si llevamos ese ejemplo a código se podría decir que la encapsulación protege la características de cada clase para que no puedan ser usadas fuera de la clase a la que pertenecen.

Bloque I - Curso de Programación Orientada a Objetos en Python, Clase 1

Programación orientada a objetos en python

**Objetivo del 2º tutorial de Curso de Programación Orientada a Objetos en Python**

Comprender como llevar a código los conceptos básicos de la POO

Habiendo explicado los que es la **POO** y sus conceptos básicos es hora de ver cómo funciona la **POO** en python y cómo interpretarlos mediante el código.

Para empezar debemos construir en código lo que sería una clase, la cual es la base para poder crear objetos que pertenezcan a esa clase, la sintaxis para construir una clase es escribiendo la palabra **class** seguida del nombre de esa clase con su primera letra en mayúscula seguido en un par de paréntesis “**()”** y dos puntos “**:**“.

class Coche():

Ahora que tenemos una clase procedemos a definir sus atributos, el comportamiento y el estado de esa clase; una clase puede tener una infinidad de atributos tantos como necesitemos. Ya que hemos declarado una clase que se refiere a un objeto de la vida real que es en este caso un coche, procedemos a implementar las siguientes propiedades:

class Coche():

largoChasis=250

anchoChasis=120

ruedas=4

enmarcha=false

Ahora que hemos definido los atributos debemos definir su comportamiento el cual viene determinado por lo que se llaman **métodos** y debemos crear uno para establecer el comportamiento, también podemos establecer infinidad de comportamientos igual que con los atributos, como hemos explicado anteriormente un comportamiento son las operaciones que puede realizar un objeto en este caso es un coche y vamos a definir el método “**arrancar**“, ya que hemos definido que uno de sus atributo es “**enmarcha = false**” que indica que no está en movimiento.

Para crear un método lo hacemos con la palabra “**def**” seguido del nombre del método que será “**arrancar**” después unos paréntesis y dentro de ellos la palabra “**self**” la cual va a hacer referencia al objeto perteneciente a la clase.

def arrancar(self):

pass

La palabra “pass” la escribimos dentro del método mientras no le asignemos ningún comportamiento ya que si no lo hacemos nuestro programa tendrá un error al ejecutarse. Ahora para declarar un objeto debemos darle un nombre (el que queramos) junto al nombre de la clase a la que queramos que pertenezca ese objeto.

miChoche=Coche()

Ahora para acceder a los atributos y a los métodos de una clase debemos definirlos escribiendo el nombre del objeto seguido de un punto “.” y el nombre del atributo o método.

Objeto y atributo:

**miCoche.largoChasis**

Objetos y método:

**miCoche.arrancar()**

Sabiendo esto, ya podemos definir un comportamiento al objeto “**miCoche**” con los siguientes métodos. Haremos que nuestro atributo “**enmarcha**” que tiene un valor “False” que nos indica que nuestro coche está detenido y le indicaremos que se ponga en marcha, para hacer esto debemos establecer dentro del método lo siguiente:

def arrancar(self):

self.enmarcha=True

Con esto estamos indicando que se cambiaran el valor predeterminado de nuestro atributo “**enmachar**” y nuestro objeto coche tendrá la capacidad de ponerse en marcha.

Ahora solo queda comprobar si nuestro objeto está en marcha o no y para eso establecemos otro método al cual podemos denominar “estado”, y debemos colocarle una condicional “if” de esta manera:

def estado(self):

if (self.enmarcha):

return "El coche está en marcha"

else:

return "El coche está detenido"

Una vez planteándonos todo esto debemos estructurar el código de esta manera:

#definir clase

class Coche():

largoChasis=250

anchoChasis=120

ruedas=4

enmarcha=False

#definir método

def arrancar(self):

#definir comportamiento

self.enmarcha=True

#definir método

def estado(self):

#definir comportamiento

if (self.enmarcha):

return "El coche está en marcha"

else:

return "El coche está detenido"

#definir objeto y a que clase pertenece

miChoche=Coche()

#imprimir lo siguiente para comprobar si funciona correctamente

print("El largo del coche es: ",miChoche.largoChasis)

print("El coche tiene ", miChoche.ruedas, " ruedas")

miChoche.arrancar()

print(miChoche.estado())

Al ejecutar esto nos mostrara la siguiente información:

El largo del coche es: 250

El coche tiene 4 ruedas

El coche está en marcha

**? Nota:**

El coche nos muestra que está en marcha porque antes de imprimir el método “**estado**” le estamos indicando que no ejecute el método “**arrancar**” que nos cambia el valor del atributo “**enmarcha**” a “**True**”, para que nos muestre que “El coche está detenido” comentamos la línea donde nos indica “**miCoche.arrancar()**” y no imprimirá esto:

El largo del coche es: 250

El coche tiene 4 ruedas

El coche está detenido

En la **POO** podemos declarar la cantidad de objetos que queramos cada uno con sus clases, atributos y comportamientos únicos que los diferencia uno de otro y lo podemos hacer de la siguiente manera:

#definir clase

class Coche():

largoChasis=250

anchoChasis=120

ruedas=4

enmarcha=False

#definir método

def arrancar(self):

#definir comportamiento

self.enmarcha=True

#definir método

def estado(self):

#definir comportamiento

if (self.enmarcha):

return "El coche está en marcha"

else:

return "El coche está detenido"

#definir objeto y a que clase pertenece

miChoche=Coche()

#imprimir lo siguiente para comprobar si funciona correctamente

print("El largo del coche es: ",miChoche.largoChasis)

print("El coche tiene ", miChoche.ruedas, " ruedas")

#miChoche.arrancar()

print(miChoche.estado())

print("---------------Segundo objeto---------------")

miChoche2=Coche()

#imprimir lo siguiente para comprobar si funciona correctamente

print("El largo del coche es: ",miChoche2.largoChasis)

print("El coche tiene ", miChoche2.ruedas, " ruedas")

miChoche2.arrancar()

print(miChoche2.estado())

Y nos imprime esto:

El largo del coche es: 250

El coche tiene 4 ruedas

El coche está detenido

---------------Segundo objeto---------------

El largo del coche es: 250

El coche tiene 4 ruedas

El coche está en marcha

Para poder plantear de mejor manera este código haremos que el método “**arrancar**” se encargue no solo de arrancar el coche sino también de informarnos cuál es el estado y que el método “**estado**” nos informe de cada una de los atributos que tiene cada objeto.

Empezamos añadiendo un nuevo parámetro al método “**arrancar**” el cual vamos a denominar “**arrancamos**”, eso lo hacemos de la siguiente manera:

def arrancar(self, arrancamos)

Y establecemos el método “arrancar” de la siguiente manera:

def arrancar(self, arrancamos):

self.enmarcha=arrancamos

if (self.enmarcha):

return "El coche está en marcha"

else:

return "El coche está detenido"

Con esto el método puede hacer la función de arrancar y de indicarnos el estado en el que el coche se encuentra, es decir, sí está en marcha o está detenido.

El método “**estado**” va a informarnos del estado de cada uno de los otros atributos que tiene el objeto y para eso hacemos lo siguiente:

def estado(self):

#definir comportamiento

print("El coche tiene ", self.ruedas, " ruedas. Un ancho de ", self.anchoChasis, " y un larde de ", self.largoChasis)

Y de esta manera el método “**arrancar**” recibe dos parámetros y el método “**estado**” nos informa del estado de actual de los atributos, ahora procedemos a mostrar en pantalla todo lo anteriormente mencionada de la siguiente manera:

miChoche=Coche()

print(miChoche.arrancar(True))

miChoche.estado()

print("---------------Segundo objeto---------------")

miChoche2=Coche()

print(miChoche2.arrancar(False))

miChoche2.estado()

Veamos el ejemplo completo y ver lo que nos muestra en pantalla:

#definir clase

class Coche():

largoChasis=250

anchoChasis=120

ruedas=4

enmarcha=False

#definir método

def arrancar(self, arrancamos):

#definir comportamiento

self.enmarcha = arrancamos

if (self.enmarcha):

return "El coche está en marcha"

else:

return "El coche está detenido"

#definir método

def estado(self):

#definir comportamiento

print("El coche tiene ", self.ruedas, " ruedas. Un ancho de ", self.anchoChasis, " y un largo dé ", self.largoChasis)

#definir objeto y a que clase pertenece

miChoche=Coche()

print(miChoche.arrancar(True))

miChoche.estado()

print("---------------Segundo objeto---------------")

miChoche2=Coche()

print(miChoche2.arrancar(False))

miChoche2.estado()

Y nos mostrara lo siguiente:

El coche está en marcha

El coche tiene 4 ruedas. Un ancho 120 y un largo de 250

---------------Segundo objeto---------------

El coche esta en detenido

El coche tiene 4 ruedas. Un ancho 120 y un largo de 250

**Estado inicial**

Con todo lo explicado anteriormente está claro que todo los objetos de la clase coche van a tener ciertos atributos en común que van a ser un ancho, largo, tener una cantidad de ruedas y que todos por defecto van a estar detenidos, a la hora de usar la programación orientada a objetos es habitual que los atributos formen parte de un estado inicial, estos quiere decir que queremos que los atributo que tenga la clase sean predeterminados, es decir, que en cuanto creemos un objeto todos sus atributos tengan un estado inicial. Para especificar el estado inicial de los atributos de una clase lo hacemos con un constructor, un constructor es un método que le da estado inicial a los objetos y esto lo hacemos de la siguiente manera:

def \_\_init\_\_(self):

self.largoChasis=250

self.anchoChasis=120

self.ruedas=4

self.enmarcha=False

Con la nomenclatura “**\_\_init\_\_**” estamos especificando que es el método constructor y va a dar un estado inicial a los atributos que pertenezcan a la clase donde lo definas. Si ejecutamos el ejemplo agregándole el constructor se ejecutara de la misma manera de antes:

#definir clase

class Coche():

def \_\_init\_\_(self):

self.largoChasis=250

self.anchoChasis=120

self.ruedas=4

self.enmarcha=False

#definir método

def arrancar(self, arrancamos):

#definir comportamiento

self.enmarcha = arrancamos

if (self.enmarcha):

return "El coche está en marcha"

else:

return "El coche está detenido"

#definir método

def estado(self):

#definir comportamiento

print("El coche tiene ", self.ruedas, " ruedas. Un ancho de ", self.anchoChasis, " y un largo de ", self.largoChasis)

#definir objeto y a que clase pertenece

miChoche=Coche()

print(miChoche.arrancar(True))

miChoche.estado()

print("---------------Segundo objeto---------------")

miChoche2=Coche()

print(miChoche2.arrancar(False))

miChoche2.estado()

Deberá mostrarnos el mismo resultado anterior:

El coche está en marcha

El coche tiene 4 ruedas. Un ancho 120 y un largo de 250

---------------Segundo objeto---------------

El coche está en detenido

El coche tiene 4 ruedas. Un ancho 120 y un largo de 250

**Encapsulación en Python**

Para entender el concepto de encapsulación a la hora de llevarlo a código de Python vamos a fijarnos bien en la secuencia de llamadas que estamos haciendo la cuál es esta:

miChoche=Coche()

print(miChoche.arrancar(True))

miChoche.estado()

print("---------------Segundo objeto---------------")

miChoche2=Coche()

print(miChoche2.arrancar(False))

miChoche2.estado()

Podemos ver que hemos creado dos objetos, al primer objeto “**miCoche**” le hemos dicho que arranque y hemos mostrado su estado, al segundo objeto “**miCoche2**” le hemos que no arranque y también hemos mostrado su estado.

Ahora si al objeto “**miCoche2**” le especificamos que tiene 2 ruedas antes de mostrar su estado al ejecutarlos nos dirá que tiene 2 ruedas, esto podemos hacerlo de la siguiente manera:

#definir clase

class Coche():

def \_\_init\_\_(self):

self.largoChasis=250

self.anchoChasis=120

self.ruedas=4

self.enmarcha=False

#definir método

def arrancar(self,arrancamos):

#definir comportamiento

self.enmarcha=arrancamos

if (self.enmarcha):

return "El coche está en marcha"

else:

return "El coche está detenido"

#definir método

def estado(self):

#definir comportamiento

print("El coche tiene ", self.ruedas, " ruedas. Un ancho de ", self.anchoChasis, " y un largo de ", self.largoChasis)

#definir objeto y a que clase pertenece

miChoche=Coche()

print(miChoche.arrancar(True))

miChoche.estado()

print("---------------Segundo objeto---------------")

miChoche2=Coche()

print(miChoche2.arrancar(False))

miChoche2.ruedas=2

miChoche2.estado()

Nos mostrara en pantalla lo siguiente:

El coche está en marcha

El coche tiene 4 ruedas. Un ancho 120 y un largo de 250

---------------Segundo objeto---------------

El coche está en detenido

El coche tiene 2 ruedas. Un ancho 120 y un largo de 250

Si utilizamos la lógica y llevamos este ejemplo a un caso de la vida real no tiene sentido un coche con 2 ruedas o más de 4 ruedas no tendría sentido y no se debería permitir que el programa ejecute algo que no tiene sentido.

Te preguntaras que es lo que se puede hacer en estos casos y es aquí donde entra el concepto de encapsulación. La encapsulación se encarga de proteger un atributo para que no se pueda modificar desde afuera de la clase, cómo pudiste ver el atributo ruedas se encuentra dentro del constructor y nosotros lo hemos modificado fuera de la clase y eso no debe ocurrir. Para evitar esto debemos encapsular ese atributo y eso lo hacemos precediendo su nombre de dos guiones bajos:

def \_\_init\_\_(self):

self.largoChasis=250

self.anchoChasis=120

self.\_\_ruedas=4 <----------------

self.enmarcha=False

Ahora estas indicando que al atributo ruedas no sea accesible desde el exterior pero si será accesible desde dentro de la clase y ahora en adelante si estas encapsulando el atributo ruedas siempre debes especificarlo con los dos guiones bajos tal y como lo declaraste dentro de la clase.

Si ejecutamos el código de la siguiente manera veras que el valor del atributo ruedas no será modificado:

#definir clase

class Coche():

def \_\_init\_\_(self):

self.largoChasis=250

self.anchoChasis=120

self.\_\_ruedas=4

self.enmarcha=False

#definir método

def arrancar(self,arrancamos):

#definir comportamiento

self.enmarcha=arrancamos

if (self.enmarcha):

return "El coche está en marcha"

else:

return "El coche está detenido"

#definir metodo

def estado(self):

#definir comportamiento

print("El coche tiene ", self.\_\_ruedas, " ruedas. Un ancho de ", self.anchoChasis, " y un larde de ", self.largoChasis)

#definir objeto y a que clase pertenece

miChoche=Coche()

print(miChoche.arrancar(True))

miChoche.estado()

print("---------------Segundo objeto---------------")

miChoche2=Coche()

print(miChoche2.arrancar(False))

miChoche2.ruedas=2

miChoche2.estado()

En pantalla se mostrara de la siguiente manera:

El coche está en marcha

El coche tiene 4 ruedas. Un ancho 120 y un largo de 250

---------------Segundo objeto---------------

El coche esta en detenido

El coche tiene 4 ruedas. Un ancho 120 y un largo de 250

Para concluir como puedes ver hemos aplicado la encapsulación para proteger el un parte del código de nuestro programa, la encapsulación puede usarse para todos los atributos y comportamiento que tiene una clase solo mientras creas que sea necesario

Bloque II - Curso de Programación Orientada a Objetos en Python, Clase 1

Herencia en Python

**Objetivo del 3º tutorial de Curso de Programación Orientada a Objetos en Python**

Comprender el significado de la herencia en la POO y como llevarla a código de Python.

Como ya sabes la **POO** se encarga de llevar las características de los objetos de la vida real a código de programación para que sea más fácil hacer aplicaciones más complejas. La herencia es igual, ella trae la función de la herencia entre persona a código de programación con algunas pequeñas diferencias, ya que en programación la herencia explica que puede crearse un objeto a partir de otro objeto ya existente. El nuevo objeto hereda todas las cualidades del objeto del que deriva y además puede añadir nuevas funcionalidades o modificar las ya existentes.

Ejemplo: Se tienen los planos de un ordenador y se quiere fabricar otro ordenador. En vez de crear uno de cero, sería mucho más sencillo basarse en el ordenador que ya se tienen y añadirle o modificarle ciertas funcionalidades como podrían ser: aumentar su capacidad, su velocidad de procesamiento, etc…

Para empezar a aplicar la herencia en código de python podemos empezar declarando una clase padre, una clase padre o súper clase es aquella que engloba todos esos atributos en común que tendrán los objetos con las clases derivadas de esa clase padre. Para declarar en código esta clase es igual que declarar una clase común pero te recomendamos que le des un nombre que identifique que esa es la clase padre.

Nosotros basaremos este ejemplo con vehículos y sus diferentes tipos para poder explicar esto con mayor facilidad, empezamos declarando la clase padre con el nombre “**Vehiculos**” y dentro de ella declaramos un constructor con su parámetro por defecto “**self**” y con dos parámetros adicionales “**marca**” y “**modelo**” dándole un estado inicial a los vehículos que hereden de esta clase una marca y un modelo:

class Vehiculos():

def \_\_init\_\_(self, marca, modelo):

Y dentro del constructor le daremos los siguientes atributos:

class Vehiculos():

def \_\_init\_\_(self, marca, modelo):

# marca es igual a la marca que le pasemos por parámetro

self.marca=marca

# modelo es igual a la modelo que le pasemos por parámetro

self.modelo=modelo

self.enmarcha=False

self.acelera=False

self.frena=False

Con esto estamos diciendo que cuando creemos un objeto de tipo vehículo u otro que herede de vehículo obligatoriamente ese objeto va tener un modelo y una marca que le tendremos que pasar al constructor y por defecto no estará en marcha, no estará acelerando y no estará frenando.

Ahora vamos a definir los comportamientos que va a tener ese objeto de la clase vehículos y los definimos de esta manera:

def arrancar(self):

self.enmarcha=True

def acelerar(self):

self.acelera=True

def frenar(self):

self.frena=True

Esto quiere decir que si llamamos a estos métodos cambiaremos el estado inicial de los atributos seleccionados.

También creamos un método de la siguiente manera:

def estado(self):

print("Marca: ", self.marca, "\nModelo: ", self.modelo, "\nEn Marcha: ", self.enmarcha, "\nAcelerando: ", self.acelera, "\nFrenando: ", self.frena)

Tal y como en ejemplos anteriores hemos creado un método que nos muestre el estado del objeto.

Ya tenemos la clase vehículos terminada, ahora tenemos que hacer que un objeto de la clase moto herede de la clase vehículos y para hacerlo tenemos que escribir el código de la siguiente manera:

class Moto(Vehiculos):

pass

Escribiendo “**Vehiculos**” dentro de los paréntesis le hemos dicho a esa clase que herede todo los atributos pertenecientes a un vehículo y hemos agregado la palabra “**pass**” debido a que no nos concentraremos en terminar de completar esa clase en este momento.

Ahora creando un objeto de clase moto veras como hereda todo los atributos y comportamientos de la clase vehículos, para crear un objeto que va a heredar lo hacemos de la siguiente manera:

miMoto=Moto("Bera", "DT")

Dentro de los paréntesis le estamos indicando la marca y el modelo ya que está heredando los atributos y comportamientos de la clase vehículos, sino hacemos esto nos mostrara un error. Ahora que hemos heredado todo ya podemos hacer que muestre el estado de la siguiente manera:

miMoto.estado()

Este es el ejemplo completo:

class Vehiculos():

def \_\_init\_\_(self, marca, modelo):

# Marca es igual a la marca que le pasemos por parámetro

self.marca=marca

# Modelo es igual a la modelo que le pasemos por parámetro

self.modelo=modelo

self.enmarcha=False

self.acelera=False

self.frena=False

def arrancar(self):

self.enmarcha=True

def acelerar(self):

self.acelera=True

def frenar(self):

self.frena=True

def estado(self):

print("Marca: ", self.marca, "\nModelo: ", self.modelo, "\nEn Marcha: ", self.enmarcha, "\nAcelerando: ", self.acelera, "\nFrenando: ", self.frena)

class Moto(Vehiculos):

pass

miMoto=Moto("Bera", "DT")

miMoto.estado()

Y esto nos muestra al ejecutarlo:

Marca: Bera

Modelo DT

En Marcha: False

Acelerando: False

Frenando: False

Gracias a la magia de la herencia en el objeto “miMoto” podemos ver que hemos llamado a un constructor con unos métodos y a simple vista pareciera que la clase moto no tiene ninguno pero como ha heredado todos lo que contiene la clase de vehículos en realidad si posee las características de un vehículo.

**Sobre escritura de métodos**

Para poder explicar la sobre escritura de métodos vamos crear un comportamiento nuevo para nuestra clase moto, como ya sabemos la clase vehículo contiene atributos generales que puede tener cualquier clase de vehículo pero una moto o cualquier otro vehículo debe por lo menos tener un atributo o comportamiento que no lo tengan los otros vehículos. Por ejemplo con una moto podemos levantarla de caballito y eso es algo que no puede hacer un avión o un tren.

Entonces procedemos a agregar el comportamiento de hacer caballito a nuestra clase moto de la siguiente manera:

class Moto(Vehiculos):

hcaballito=""

def caballito(self):

self.hcaballito = "Voy haciendo caballito"

Primero declaramos un atributo y lo llamamos “hcaballito” con un valor vacío entre sus comillas y definimos un método que nos indique que está haciendo caballito. Ya con esto si hacemos una llamada al método para levantar la moto de caballito, le mostraremos el ejemplo completo:

class Vehiculos():

def \_\_init\_\_(self, marca, modelo):

# Marca es igual a la marca que le pasemos por parámetro

self.marca=marca

# Modelo es igual a la modelo que le pasemos por parámetro

self.modelo=modelo

self.enmarcha=False

self.acelera=False

self.frena=False

def arrancar(self):

self.enmarcha=True

def acelerar(self):

self.acelera=True

def frenar(self):

self.frena=True

def estado(self):

print("Marca: ", self.marca, "\nModelo: ", self.modelo, "\nEn Marcha: ", self.enmarcha, "\nAcelerando: ", self.acelera, "\nFrenando: ", self.frena)

class Moto(Vehiculos):

hcaballito=""

def caballito(self):

self.hcaballito="Voy haciendo caballito"

miMoto=Moto("Bera", "DT")

miMoto.caballito()

miMoto.estado()

Si ejecutamos este ejemplo “funcionara” no mostrara ningún error pero no nos mostrara si está haciendo caballito o no:

Marca: Bera

Modelo DT

En Marcha: False

Acelerando: False

Frenando: False

Esto se debe a que el método estado que estamos heredando de la clase vehículos no tiene la función de hacer caballito pero no podemos declarar la función de hacer caballito al método estado de la clase vehículos ya que terminaría siendo una función que cualquier vehículo pueda realizar pero un avión o una lancha no pueden hacer semejante acción y sería una clase mal declarada.

Para solucionar este problema debemos hacer que la clase moto tenga su propio método de estado con las mismas funciones y agregando al método estado de la clase moto la función de hacer caballito:

class Vehiculos():

def \_\_init\_\_(self, marca, modelo):

# Marca es igual a la marca que le pasemos por parámetro

self.marca=marca

# Modelo es igual a la modelo que le pasemos por parámetro

self.modelo=modelo

self.enmarcha=False

self.acelera=False

self.frena=False

def arrancar(self):

self.enmarcha=True

def acelerar(self):

self.acelera=True

def frenar(self):

self.frena=True

def estado(self):

print("Marca: ", self.marca, "\nModelo: ", self.modelo, "\nEn Marcha: ", self.enmarcha, "\nAcelerando: ", self.acelera, "\nFrenando: ", self.frena)

class Moto(Vehiculos):

hcaballito=""

def caballito(self):

self.hcaballito="Voy haciendo caballito"

def estado(self):

print("Marca: ", self.marca, "\nModelo: ", self.modelo, "\nEn Marcha: ", self.enmarcha, "\nAcelerando: ", self.acelera, "\nFrenando: ", self.frena, "\n", self.hcaballito)

miMoto=Moto("Bera", "DT")

miMoto.caballito()

miMoto.estado()

Aquí surge un dilema debido que al momento de declarar un método igual que el de la clase que estamos heredando te preguntaras que método estamos llamando en la línea “**miMoto.estado()**” si es el de la clase moto o la clase vehículos, pues en realidad estamos llamando al método de la clase moto, ya que, cuando se presentan estos casos siempre se sobre escribe el método que estamos heredando de la clase padre y esto invalida, anula o sobrescribe el método de la clase padre de tal forma que ahora si nos mostrará en pantalla que la moto está haciendo caballito:

Marca: Bera

Modelo: DT

En Marcha: False

Acelerando: False

Frenando: False

Voy haciendo caballito

Para concluir cabe resaltar que hacer que la moto haga caballito es un método totalmente opcional, si comentamos lo línea “miMoto.caballito()” nos mostrara los datos heredados de la clase vehículo con total normalidad ya que dentro del método caballito declaramos “hcaballito” con un valor vacío así que el método estado de la clase moto no mostrara nada si no lo está haciendo.

**Herencia Múltiple**

Para empezar debemos darle a nuestro programa soporte para vehículos eléctricos, para esto como ya debes saber lo hacemos declarando una clase que tenga todos los atributos y comportamiento que tiene un vehículo de este tipo, así de esta manera:

class VElectricos():

def \_\_init\_\_(self):

self.autonomia=100

def cargarEnergia(self):

self.cargando=True

Ahora después de la línea, de la última instrucción declaramos otra clase que llamaremos “**BecicletaElectrica**” y esta se va a encarga construir los vehículos de tipo eléctricos. Aquí te puedes dar cuenta que una bicicleta eléctrica posee todos los atributos y comportamientos de un vehículo y todo lo que posee un vehículo eléctrico, pues es aquí donde se aplica el tema de la herencia múltiple la cual es donde declaramos un clase que heredé los atributo y comportamientos de más de una clase padre. Para especificar una herencia múltiple debemos separar los nombres de las clases heredadas con un coma “**,**” dentro de los paréntesis de la clase, así de esta manera:

class BicicletaElectrica(VElectricos,Vehiculos):

pass

Con esto estamos diciendo que una bicicleta eléctrica hereda de “**VElectricos**” y “**Vehiculos**”, por ahora no le indicaremos nada a esta clase y por eso escribimos la palabra “**pass**”.

Ya podemos crear un objeto con la clase bicicleta eléctrica pero tenemos un dilema ya que ambas clases tiene un constructor y te preguntaras cual está heredando, si nosotros especificamos una marca y un modelo nos dará error:

miBici=BicicletaElectrica(“Ordea”, “H310”)

Nos dirá que el constructor toma un argumento posicional pero se le han dado tres y esto sucede porque cuando hay herencia múltiple se le da preferencia a la primera clase heredada que escribes dentro de los paréntesis que en este caso fue “**VElectricos**” y por lo tanto toma como constructor al que le corresponde a esa clase o si hay atributos y comportamiento del mismo nombre en varias clases siempre tomara en cuenta todos lo que estén en la primera que le especificamos.

Seguro te preguntaras si puede darle una marca y un modelo al objeto “**miBici**” y por supuesto que puedes hacerlo y para hacer eso debes usar la función “**super()**” la cual es la que va llamar a los métodos de la clase padre a las otras clases.

Para empezar nos vamos a concentrar en la clase “**VElectricos**” y en el constructor de esa clase vamos a especificar antes de “**self.autonomia**” la función “**super()**” que llamara al constructor de la clase vehículos y pasarle, una marca y un modelo, y que no se les olvide especificarle que herede de vehículos así de esta manera:

class VElectricos(Vehiculos):

def \_\_init\_\_(self, marca, modelo):

super().\_\_init\_\_(marca, modelo)

self.autonomia=100

def cargarEnergia(self):

self.cargando=True

Ahora veamos en ejemplo completo y que nos mostrara al ejecutarlo:

class Vehiculos():

def \_\_init\_\_(self, marca, modelo):

# Marca es igual a la marca que le pasemos por parámetro

self.marca=marca

# Modelo es igual a la modelo que le pasemos por parámetro

self.modelo=modelo

self.enmarcha=False

self.acelera=False

self.frena=False

def arrancar(self):

self.enmarcha=True

def acelerar(self):

self.acelera=True

def frenar(self):

self.frena=True

def estado(self):

print("Marca: ", self.marca, "\nModelo: ", self.modelo, "\nEn Marcha: ", self.enmarcha, "\nAcelerando: ", self.acelera, "\nFrenando: ", self.frena)

class Moto(Vehiculos):

hcaballito=""

def caballito(self):

self.hcaballito="Voy haciendo caballito"

def estado(self):

print("Marca: ", self.marca, "\nModelo: ", self.modelo, "\nEn Marcha: ", self.enmarcha, "\nAcelerando: ", self.acelera, "\nFrenando: ", self.frena, "\n", self.hcaballito)

class VElectricos(Vehiculos):

def \_\_init\_\_(self, marca, modelo):

super().\_\_init\_\_(marca, modelo)

self.autonomia=100

def cargarEnergia(self):

self.cargando=True

miMoto=Moto("Bera", "DT")

miMoto.caballito()

miMoto.estado()

class BicicletaElectrica(VElectricos,Vehiculos):

pass

miBici=BicicletaElectrica("Orbea", "H310")

miBici.estado()

Y Al ejecutarlo nos muestra esto:

Marca: Bera

Modelo DT

En Marcha: False

Acelerando: False

Frenando: False

Voy haciendo caballito

Marca: Orbea

Modelo H310

En Marcha: False

Acelerando: False

Frenando: False

Como puedes ver una puede heredar de todas las que quiera mientras sea necesario.

Bloque II - Curso de Programación Orientada a Objetos en Python, Clase 2

Polimorfismo en Python

**Objetivo del 4º tutorial de Curso de Programación Orientada a Objetos en Python**

Comprender el significado del polimorfismo en la POO y como llevarla a código de Python.

Para aquellas personas que ven por primera vez la programación orientada a objetos debe ser un poco confuso en término de polimorfismo y más por lo estrambótico de la palabra la cual viene del griego Poli que significa muchas y morfismo que significa formas, es decir, muchas formas.

En programación se da el concepto de polimorfismo cuando un objeto puede cambiar de forma dependiendo del contexto donde se utilice y por lo tanto ese objeto también cambia de comportamiento. Si se da de manera como ejemplos anteriores donde un objeto puede pasar de ser un camión a una moto o un carro por lo tanto ese objeto cambiaria de comportamiento dependiendo del contexto donde se utilice.

Es una característica potente a la hora de usarse en python, ya que python es un lenguaje de estructura dinámica lo cual facilita su uso al momento de programarlo, es decir, que no necesitamos especificar de qué tipo es el objeto al contrario de lo que ocurre en otros lenguajes como java.

Nuestro ejemplo en python empieza declarando una clase que llamaremos “**Coche**”, esta clase va a tener un método el cual va a ser “**desplazamiento**” y nos va imprimir un texto que nos diga “**Me desplazo utilizando cuatro ruedas**” así de esta manera:

class Coche():

def desplazamiento(self):

print("Me desplazo utilizando cuatro ruedas")

Ahora declaramos una clase que llamaremos “**Moto**” con un método igual de desplazamiento y procederemos a hacer que imprima “**Me desplazo utilizando dos ruedas**” así de esta manera:

class Moto():

def desplazamiento(self):

print("Me desplazo utilizando dos ruedas")

También declararemos una clase que se llamara camión de la mismo manera que las otras dos pero esta se desplazara utilizando seis ruadas:

class Camion():

def desplazamiento(self):

print("Me desplazo utilizando seis ruedas")

Ya que tenemos las clases creamos los siguientes objetos con las clases que ya hemos creado y que impriman los métodos de desplazamiento:

miVehiculo=Moto()

miVehiculo.desplazamiento()

miVehiculo2=Coche()

miVehiculo2.desplazamiento()

miVehiculo3.Camion()

miVehiculo3.desplazamiento()

Este es el ejemplo completo:

class Coche():

def desplazamiento(self):

print("Me desplazo utilizando cuatro ruedas")

class Moto():

def desplazamiento(self):

print("Me desplazo utilizando dos ruedas")

class Camion():

def desplazamiento(self):

print("Me desplazo utilizando seis ruedas")

miVehiculo=Moto()

miVehiculo.desplazamiento()

miVehiculo2=Coche()

miVehiculo2.desplazamiento()

miVehiculo3=Camion()

miVehiculo3.desplazamiento()

Si ejecutamos el ejemplo completo nos mostrara lo siguiente:

Me desplazo utilizando dos ruedas

Me desplazo utilizando cuatro ruedas

Me desplazo utilizando seis ruedas

El polimorfismo nos permite prescindir de todo estos objetos y declarar una función que podemos llamar “**desplazamientoVehiculo**” y esta función va a recibir un objeto por parámetro que podemos llamar “**vehiculo**”. Esta función lo que va a hacer es utilizar ese objeto que recibe por parámetro para llamar al método desplazamiento:

def desplazamientoVehiculo(vehiculo):

vehiculo.desplazamiento()

Es aquí donde empieza la magia del polimorfismo creando un objeto de la clase camión:

miVehiculo=Camion()

Y ahora hago una llamada a la función “**desplazamientoVehiculo**” y le pasamos por parámetro el objeto de tipo camión que acabamos de crear así de esta manera:

desplazamientoVehiculo(miVehiculo)

Con esto ya estamos aplicando el polimorfismo debido a que cuando le pasamos por parámetro el objeto de tipo camión el polimorfismo hace que la función “**desplazamientoVehiculo**” detecte que el objeto es de tipo camión y nos llama al método desplazamiento de la clase camión.

Ejemplo completo:

class Coche():

def desplazamiento(self):

print("Me desplazo utilizando cuatro ruedas")

class Moto():

def desplazamiento(self):

print("Me desplazo utilizando dos ruedas")

class Camion():

def desplazamiento(self):

print("Me desplazo utilizando seis ruedas")

def desplazamientoVehiculo(vehiculo):

vehiculo.desplazamiento()

miVehiculo=Camion()

desplazamientoVehiculo(miVehiculo)

Si ejecutamos el ejemplo nos muestra lo siguiente:

Me desplazo utilizando seis ruedas

Como tarea para practicar te recomendamos que intentes hacerlo con las otras clases que hemos declarado y así ver la magia del polimorfismo.

Bloque III - Curso de Programación Orientada a Objetos en Python, Clase 1

Sobrecarga de métodos en la POO con Python

**Objetivo del 5º tutorial de Curso de Programación Orientada a Objetos en Python**

Conocer la lista de los operadores aritméticos que podemos usar con la sobrecarga de métodos.

Mostrar el uso del método “\_\_str\_\_” para imprimir un objeto.

Usar la sobrecarga de operadores aritméticos.

La sobrecarga de métodos es una técnica muy poderosa de la POO ya que nos permite usar los operadores aritméticos con nuestras propias clases, es decir, podemos usar la suma, la resta, la multiplicación, la división o los operadores de comparación como mayor que, menor que y operadores booleanos en nuestras clases proporcionando la suma entre objetos o la comparación entre ellos.

En python se simplifica mucho llevar a cabo la sobrecarga de métodos debido a que python tiene una serie de métodos que corresponden a cada operador:

+ \_\_add\_\_

– \_\_sub\_\_

\* \_\_mul\_\_

\*\* \_\_pow\_\_

/ \_\_truediv\_\_

// \_\_floordiv\_\_

% \_\_mod\_\_

<< \_\_lshift\_\_

>> \_\_rshift\_\_

& \_\_and\_\_

| \_\_or\_\_

^ \_\_xor\_\_

~ \_\_invert\_\_

< \_\_lt\_\_

<= \_\_le\_\_

== \_\_eq\_\_

!= \_\_ne\_\_

> \_\_gt\_\_

>= \_\_ge\_\_

Con esto solo debes saber cuál es el método que corresponde a cada operador para poder usarlos.

En este tutorial veremos el ejemplo de una sobrecarga para una clase de tipo vector y haremos algunas operaciones con ese vector para ver la simplicidad de este tipo de código.

Para empezar debemos importar el modulo “**math**” ya que vamos a hacer uso de una raíz cuadre (importar a “**math**” no es necesario para realizar las sobrecargas). Ahora declaramos la clase vector con dos parámetro que llamaremos “**px=0**” y “**py=0**” de esta manera:

import math

class Vector:

def \_\_init\_\_(self, px=0, py=0):

self.x=px

self.y=py

El siguiente método que debemos crear es un método **“\_\_str\_\_**” cuya función es indicar como queremos que se muestre esa clase en particular regresando en una cadena con el formato correspondiente:

def \_\_str\_\_(self):

return "(%f,%f)"%(self.x, self.y)

Este “**return**” está queriendo decir que nuestros componentes “**x**” y “**y**” se muestren dentro de un paréntesis.

Ahora empezaremos la primera sobrecarga la cual haremos con el operador “**+**”. Para sobrecargar el operador “**+**” debemos tomar en cuenta que para realizar una suma se necesitan dos valores, uno que este a la izquierda antes del “**+**” y otra a la derecha después del signo “**+**”, esto lo hacemos de la siguiente manera:

def \_\_add\_\_(self, operando):

mx=self.x+operando.x

my=self.y+operando.y

return Vector(mx,my)

El parámetro “**operando**” hace referencia al objeto que va a estar a la derecha del “**+**” y “**self**” al que está a la izquierda de signo “**+**”. Dentro de este método se crearon dos variable temporales “**mx**” y “**my**”, con valores que indican que “**self.x**” y “**self.y**” son el operando que se encuentra en la izquierda, después “**operando.x**” y “**operando.y**” son el operando que está a la derecha. Ya con esto escribimos un “**return**” que nos regresara los valores calculados en un vector, regresar el resultado en un objeto de tipo vector es necesario porque en realidad estamos haciendo la suma de dos vectores.

**? Nota:**

Cuando ustedes realicen sus propias sobrecargas y lleven a cabo un método como “**\_\_add\_\_**” dentro del método hay que colocar la suma de dos objetos deacuerdo a las propias reglas de estos objetos que ustedes estén creando ya que en nuestro ejemplo lo hacemos con un objeto de tipo vector.

Ahora vamos a llevar acabo la sobrecarga de operadores de comparación, para esto debemos crear un método que llamaremos “**magnitud**” ya que vamos a comparar la magnitud de los dos vectores que se están sumando, eso lo hacemos de esta manera:

def magnitud(self):

return math.sqrt(self.x\*\*2 + self.y\*\*2)

Este método compara nuestros vectores y nos devuelve el valor de su magnitud, esto lo hace usando “**math**” y “**.sqrt**” para sacar la raíz cuadrada y dentro de sus paréntesis hacemos la operación de la raíz cuadrada de “**x**” y “**y**”.

Aprovechando este método procedemos a hacer la sobrecarga de operador de igualdad “**==**” con el método “**\_\_eq\_\_**”, el operador igualdad también tiene dos operando que son el izquierdo “**self**” como primer parámetro y el operando derecho va a ser el que va a entrar como segundo parámetro, dentro del método simplemente se hace la comparación:

def \_\_eq\_\_(self, operando):

return self.magnitud()==operando.magnitud()

Ahora haremos una sobrecarga con el operador mayor que “**>**” usando el método “**\_\_gt\_\_**” de la misma manera que el método para comparar los operadores:

def \_\_gt\_\_(self, operando):

return self.magnitud()>operando.magnitud()

Ahora creamos nuestros objetos de tipo vector con los siguientes valores:

a=Vector(5,3)

b=Vector(-1,2)

d=Vector(5,3)

Y por último vamos a imprimir los objetos “**a**” y “**b**”:

print(a)

print(b)

Te mostraremos el ejemplo completo y su resultado en pantalla:

import math

class Vector:

def \_\_init\_\_(self, px=0, py=0):

self.x=px

self.y=py

def \_\_str\_\_(self):

return "(%f,%f)"%(self.x, self.y)

def \_\_add\_\_(self, operando):

mx=self.x+operando.x

my=self.y+operando.y

return Vector(mx,my)

def magnitud(self):

return math.sqrt(self.x\*\*2 + self.y\*\*2)

def \_\_eq\_\_(self, operando):

return self.magnitud()==operando.magnitud()

def \_\_gt\_\_(self, operando):

return self.magnitud()>operando.magnitud()

a=Vector(5,3)

b=Vector(-1,2)

d=Vector(5,3)

print(a)

print(b)

Nos mostrara lo siguiente:

(5.000000,3.000000)

(-1.000000,2.000000)

El programa nos imprime el resultado de esta manera ya que hemos creado el método “**\_\_str\_\_**” que nos retornara los valores de esta manera sin necesidad de especificarle ningún método o haciendo referencia a ningún parámetro.

Ahora que hemos mostrado que el programa funciona vamos a seguir completando el ejemplo creando un objeto llamado “**c**” que tenga como valor la suma de “**a+b**”, de esta forma:

c=a+b

Es aquí donde empieza la magia de la sobrecarga de métodos ya que con esto hemos llamado al método “**\_\_add\_\_**” sin necesidad de hacer cosas como “c.vector.(a+b)” o cosas por el estilo simplificando el mantenimiento del código y la reparación de errores en el programa ya que al colocar “**a**” a la izquierda del signo “**+**” lo estamos especificando como el parámetro “**selt**” del método “**\_\_add\_\_**” y “**b**” que está a la derecha del signo “**+**” recibe el parámetro “**operando**”.

Agregando esto procedemos a probar todo esto imprimiendo  “**c**” y probar el método de magnitud de esta manera:

import math

class Vector:

def \_\_init\_\_(self, px=0, py=0):

self.x=px

self.y=py

def \_\_str\_\_(self):

return "(%f,%f)"%(self.x, self.y)

def \_\_add\_\_(self, operando):

mx=self.x+operando.x

my=self.y+operando.y

return Vector(mx,my)

def magnitud(self):

return math.sqrt(self.x\*\*2 + self.y\*\*2)

def \_\_eq\_\_(self, operando):

return self.magnitud()==operando.magnitud()

def \_\_gt\_\_(self, operando):

return self.magnitud()>operando.magnitud()

a=Vector(5,3)

b=Vector(-1,2)

d=Vector(5,3)

print(a)

print(b)

c=a+b

print(c)

print(a.magnitud())

Y nos mostrara esto al ejecutarlo:

(5.000000,3.000000)

(-1.000000,2.000000)

(4.000000,5.000000)

5.830951894845301

Como puede observar nos imprime “**a**” y “**b**”, después la suma de los dos, es decir, “**c**” y la magnitud de “**a**”.

Ahora solo falto hacer las comparaciones y para esto hacemos uso del condicional “**if**” de esta manera:

if(a==b):

print("a y b son iguales")

else:

print("a y b no son iguales")

if(a==d):

print("a y d son iguales")

else:

print("a y d no son iguales")

if(a>b):

print("a es mayor")

else:

print("b es mayor")

Como puedes ver para usar la sobrecargar de métodos solo tenemos que hacer uso del símbolo aritmético que representa al método sobrecargar que se crea dentro de una clase, te mostraremos el ejemplo completo y lo que nos muestra al ejecutarlo:

import math

class Vector:

def \_\_init\_\_(self, px=0, py=0):

self.x=px

self.y=py

def \_\_str\_\_(self):

return "(%f,%f)"%(self.x, self.y)

def \_\_add\_\_(self, operando):

mx=self.x+operando.x

my=self.y+operando.y

return Vector(mx,my)

def magnitud(self):

return math.sqrt(self.x\*\*2 + self.y\*\*2)

def \_\_eq\_\_(self, operando):

return self.magnitud()==operando.magnitud()

def \_\_gt\_\_(self, operando):

return self.magnitud()>operando.magnitud()

a=Vector(5,3)

b=Vector(-1,2)

d=Vector(5,3)

print(a)

print(b)

c=a+b

print(c)

print(a.magnitud())

if(a==b):

print("a y b son iguales")

else:

print("a y b no son iguales")

if(a==d):

print("a y d son iguales")

else:

print("a y d no son iguales")

if(a>b):

print("a es mayor")

else:

print("b es mayor")

Al ejecutarlo nos muestra lo siguiente:

(5.000000,3.000000)

(-1.000000,2.000000)

(4.000000,5.000000)

5.830951894845301

a y b no son iguales

a y d son iguales

a es mayor

Este ejemplo nos muestra claramente la utilidad de la sobrecarga de métodos la cual permite que los operadores aritméticos formen parte de las clases donde los hayas creado y que pueden ser usados o manejados por los objetos que pertenezcan a la clase con métodos de sobrecarga.

Bloque III - Curso de Programación Orientada a Objetos en Python, Clase 2

Método de fábrica POO en Python

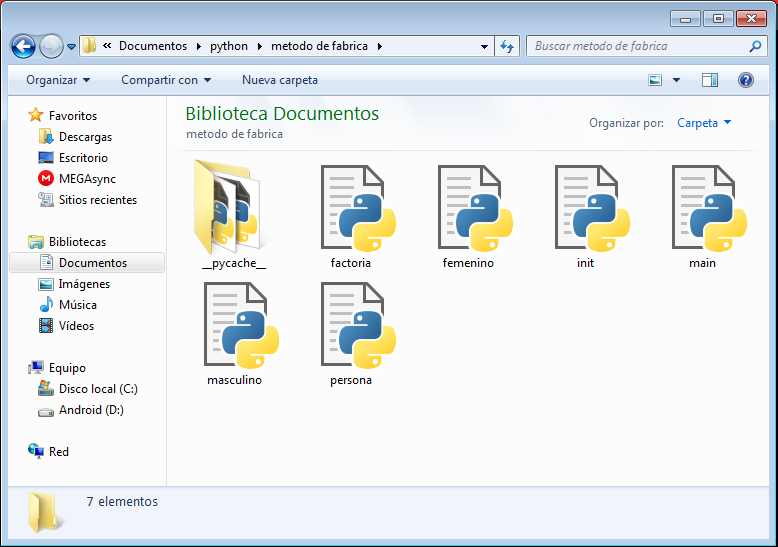
**Objetivo del 6º tutorial de Curso de Programación Orientada a Objetos en Python**

Dar a conocer como se utilizan los métodos de fabrica de la POO en Python

Los métodos de fábrica son parte de los patrones de creación, patrones de comportamiento y patrones de estructuración. Como su nombre lo indica estos métodos se encargan de fabricar objetos de diferentes tipos, esta técnica es muy útil cuando no se sabe que objetos se deben crear y debido a esto los objetos se crearan en el momento que se está ejecutando el programa. Los métodos de fábrica deben ser creados con ciertos parámetros que indiquen que objetos debe crear según sea la manera que lo hagamos al escribir el código de dichos métodos.

Para empezar con nuestro ejemplo te dejaremos claro que haremos un programa que creara personas.

Este ejemplo tendrá una súper clase que se llamara “**Personas**”, dos clases que heredaran de persona que se llamaran “**Masculinos**” y “**Femenino**”, una clase que se llame “**Factoría**” que se encargara de fabricar los objetos y todo este código se dividirá en diferente ficheros siendo “**main.py**” donde se iniciara la aplicación tal y como se muestra en esta imagen:



Dentro del fichero “**persona.py**” escribiremos nuestra clase padre “**Persona**” de la siguiente manera:

"""

Clase que define a una persona

"""

class Persona(object):

"""Para nuestro caso, una persona tendra un nombre,

una edad y un genero, por lo general

en Java esta clase sería una 'interfaz' """

def \_\_init\_\_(self):

self.nombre = None

self.edad = None

self.genero = None

# Algunos getters ...

def get\_nombre(self):

return self.nombre

def get\_edad(self):

return self.edad

def get\_genero(self):

return self.genero

def \_\_str\_\_(self):

return "Informacion de una persona:\n1. Nombre: {n}\n2. Edad: {e}\n3. Genero: {g}".format(n=self.get\_nombre(), e=self.get\_edad(), g=self.get\_genero())

Dentro del fichero “**femenino.py**” crearemos la clase “**Femenino**” que herede de persona haciendo una llamada al fichero “**Persona.py**” de esta manera:

from persona import Persona

class Femenino(Persona):

"""Esta clase hereda de la super clase Persona,

solo definiremos su constructor"""

def \_\_init\_\_(self, nombre, edad, genero):

self.nombre = nombre

self.edad = edad

self.genero = genero

print ("Hola Miss "+nombre+" su edad es "+str(edad))

Dentro del fichero “**masculino.py**” crearemos la clase “**Masculino**” que herede de persona haciendo una llamada al fichero “**Persona.py**” de esta manera:

from persona import Persona

class Masculino(Persona): # Heredamos de Persona

"""Esta clase hereda de la super clase Persona,

solo definiremos su constructor"""

def \_\_init\_\_(self, nombre, edad, genero):

self.nombre = nombre

self.edad = edad

self.genero = genero

print ("Hola mister "+nombre+" su edad es "+str(edad))

Dentro del fichero “**factoria.py**” crearemos nuestro método de fábrica que se encargara de crear los objetos pertenecientes a la clase masculino o a la clase femenino dependiendo del que queramos crear pasándole por parámetro “**M**” si es masculino y “**F**” si es femenino importando los ficheros “**masculino.py**” y “**femenino.py**”:

from femenino import Femenino

from masculino import Masculino

class Factoria(object):

"""Esta clase es nuestra factoria, como ya sabes

Python define un constructor sin argumentos

por default para cada clase, por eso no hace falta

escribir uno"""

def get\_persona(self, nombre, genero, edad):

#genero es el parametro usado por la factoria

#para elegir el obj a crear

if (genero is 'F'):

return Femenino(nombre, edad, genero)

elif (genero is 'M'):

return Masculino(nombre, edad, genero)

Y por último dentro de “**main.py**” importaremos al fichero “**factoria.py**” donde de mostrar como factoría crea los objetos de la siguiente manera:

import factoria

if \_\_name\_\_ == '\_\_main\_\_':

mi\_factoria = factoria.Factoria()

#Factoria, crea a una persona!

persona = mi\_factoria.get\_persona('Guido Vann Rosum', 'M', 30)

#se ha creado una persona masculina

print (persona)

# print persona.get\_nombre()

# print persona.get\_genero()

Como puedes observar en la clase “**Persona**” se ha aplicado la sobrescritura en el método “**\_\_str\_\_**” para que nos muestre un mensaje personalizado, en el fichero “**main.py**”, cuando escribimos **print (persona)**, estamos usando el método “**\_\_str\_\_**” (que hemos sobrescrito) para mostrar información del objeto y si ejecutamos el ejemplo usando el fichero “**main.py**” nos mostrara lo siguiente:

Hola mister Guido Vann Rosum su edad es de 30

Informacion de una persona

1. Nombre: Guido Vann Rosum

2. Edad: 30

3. Genero: M

Como puedes ver se ha creado un objeto Persona, específicamente de género Masculino debido a que hemos pasado por parámetro el carácter ‘M’ al constructor, por ello la fábrica ha decidido crear un objeto Masculino.

Puede probar creando una persona de la clase femenino para que pruebes la funcionalidad de este tipo de algoritmos, también puede agregar más método y las clases que tú quieras.

Bloque III - Curso de Programación Orientada a Objetos en Python, Clase 3

Clases internas POO en Python

**Objetivo del 7º tutorial de Curso de Programación Orientada a Objetos en Python**

Indagar en el concepto de clases internas.

Las clases internas o clases anidadas son aquellas clases que se crean dentro de la estructura de una clase al igual que los métodos y los atributos. Una clase puede tener varias clases internas tantas como sea necesario.

**Ejemplo de clases internas**

Para empezar crearemos una clase que se llame “**Humano**”, dentro de esta clase creamos un constructor y dentro de ese constructor le daremos un nombre a el objeto que pertenezca a esa clase junto a la capacidad de hablar especificando que esa capacidad la otorga una clases dentro de la misma que llamaremos “**Cabeza**” con un método que le permita hablar a nuestro objeto perteneciente a esa clase.

class Humano:

def \_\_init\_\_(self):

self.nombre = 'Guido'

self.cabeza = self.Cabeza()

class Cabeza:

def hablar(self):

return 'hablando...’

Ya teniendo todo esto procedemos a crear el objeto e imprimir si métodos de la siguiente manera:

if \_\_name\_\_ == '\_\_main\_\_':

guido = Humano()

print (guido.nombre)

print (guido.cabeza.hablar())

El ejemplo completo debe ser algo como esto:

class Humano:

def \_\_init\_\_(self):

self.nombre = 'Guido'

self.cabeza = self.Cabeza()

class Cabeza:

def hablar(self):

return 'hablando...'

if \_\_name\_\_ == '\_\_main\_\_':

guido = Humano()

print (guido.nombre)

print (guido.cabeza.hablar())

Y al ejecutarlo nos mostrara lo siguiente:

Guido

hablando...

**Ejemplo de múltiples clases internas**

Como mencionamos al principio una clase puede tener infinita cantidad de clases internas tantas como sean necesaria, como en este ejemplo:

class Humano:

def \_\_init\_\_(self):

self.nombre = 'Guido'

self.cabeza = self.Cabeza()

self.cerebro = self.Cerebro()

class Cabeza:

def hablar(self):

return 'hablando...'

class Cerebro:

def pensar(self):

return 'pensando...'

if \_\_name\_\_ == '\_\_main\_\_':

guido = Humano()

print (guido.nombre)

print (guido.cabeza.hablar())

print (guido.cerebro.pensar())

Si ejecutas el ejemplo te mostrara lo siguiente:

Guido

hablando...

pensando…

Esta es la forma de crear clases internas, al usar clases internas puedes hacer que tu código sea aún más orientado a objetos y ayuda a estructurar mejor el código de tus programas.

Prueba final del Curso de Programación Orientada a Objetos en Python

Principio del formulario

¿La POO no permite la reutilización de código?

Verdadero

Falso

¿Cuáles de estos lenguajes de programación son orientados a objetos?

Fortran, Cobol, Basic

C++, Java, Visual.NET

¿Cuál es la forma más fácil de programar?

Con la programación orientada a objetos

Con la programación orientada a procedimientos

¿La programación orientada a objetos se basa en traer objetos de la vida real a código?

Verdadero

Falso

¿Un objeto no necesita pertenecer a una clase?

Verdadero

Falso

¿Una clase puede tener atributos, comportamientos y estados?

Verdadero

Falso

¿La POO no es modularizada?

Verdadero

Falso

¿La encapsulación protege atributos y comportamientos para que no puedan ser usados fuera de la clase?

Verdadero

Falso

¿En la POO si existe un fallo en alguna línea de código el programa aun así podrá ejecutarse?

Verdadero

Falso

¿Un objeto puede heredar de cualquier clase?

Verdadero

Falso

¿Cuál de estos dos ejemplos son la sintaxis para declarar una clase?

class NombredelaClase():

def class nomdelaClase(if):

¿Cómo hacemos una llamada a un método para que se muestre al ejecutar el programa?

miObjeto.metodo()

miObjeto=metoto()

¿La herencia permite que una clase posea los atributos y comportamientos de otras sin haberlos declarado dentro de la clase que está heredando?

Verdadero

Falso

En código ¿Cómo hacemos para que una clase herede de otra?

class Coche(Vehiculo):

class Coche()=def Vehiculo(self):

¿Un objeto es un ejemplar perteneciente a una clase?

Verdadero

Falso

¿En la herencia es conveniente crear un clase padre?

Verdadero

Falso

¿La modularización hace que todas las clases trabajen juntas? y si una clase falla ¿Fallan todas?

Verdadero

Falso

¿La POO permite facilitar la depuración de código?

Verdadero

Falso

¿Polimorfismo es cuando un objeto puede cambiar de forma dependiendo del contexto donde se utilice y por lo tanto ese objeto también cambia de comportamiento?

Verdadero

Falso

¿Un Atributo o comportamiento puede ser cambiado fuera de la clase mientras no esté encapsulado?

Verdadero

Falso

Final del formulario