TEXT CLASS REVIEW

TEMAS A TRATAR EN EL CUE

- Creación de una Clase en Python.
- Definición de Atributos.
- Definición de Métodos.
- Método Constructor.
- Métodos Accesadores y Mutadores.
- Colaboración y Composición.
- Sobrecarga de Métodos.

La Programación Orientada a Objetos es una forma de programar que trata de encontrar una solución a estos problemas.

Una clase es el elemento esencial de la POO, que define una serie de elementos que determinan un estado (datos) y un comportamiento (operaciones sobre los datos que modifican su estado).

Las propiedades y comportamiento de un tipo de objeto concreto es una clase. La instanciación es la lectura de estas definiciones y la creación de un objeto a partir de ellas, que se conoce como instancia de una clase.

CREACIÓN DE UNA CLASE EN PYTHON

Dentro del lenguaje de programación Python, la POO se representa de una manera simple y fácil de escribir, es importante tener los conceptos básicos teóricos y funcionales relacionados a la POO para poder plasmarlo en el código.

La teoría de la POO nos indica que todos los objetos deben pertenecer a una clase, ya que ésta es la base para diferenciarse unos de otros, teniendo atributos y comportamientos que los distingan de otros objetos que pertenezcan a otras clases. Para crear clases en Python lo hacemos de la siguiente manera:

```
1 class Vehiculo():
```

Como se puede ver, para crear una clase vehículo lo hacemos escribiendo la palabra reservada **class**, seguida del nombre de la clase y un par de paréntesis. Se debe tener en cuenta que el nombre de la clase que se va a crear debe empezar por mayúsculas, y que si tiene más de una palabra se debe usar la notación CamelCase, o notación de camello.

DEFINICIÓN DE ATRIBUTOS

Ya tenemos una clase, debemos definir sus atributos y comportamientos. Para ello se deja la sangría o indentación de 4 espacios correspondiente para indicarle que estamos escribiendo dentro de la clase, y para definir un atributo simplemente creamos una variable con total normalidad, asignándole un valor de ser necesario por defecto.

```
1 class Vehiculo():
2     ruedas = 4
```

DEFINICIÓN DE MÉTODOS O COMPORTAMIENTO

Para crear los comportamientos de una clase en Python, también conocidos como métodos, se definen tal cual como la definición de una función con la palabra por defecto **def**, y el nombre de dicho método; pero, para diferenciar un método de una función, lo hacemos escribiendo dentro de sus paréntesis el parámetro **self**:

```
1 class Vehiculo():
2     ruedas=4
3 
4    def desplazamiento(self):
5         pass
```

La palabra **self** hace referencia a los objetos que pertenezcan a la clase, y la palabra **pass** que colocamos dentro del método, le indica al intérprete de Python que todavía no le hemos definido ningún funcionamiento a ese método, ya que, si no escribimos la palabra **pass** cuando todavía no le asignemos nada al método, al ejecutarlo nos dará un error.

Para definir un método dentro de la clase `Vehiculo()` llamado desplazamiento, y que nos muestre un `print` que diga "El vehículo se está desplazando sobre 4 ruedas", escribimos:

```
class Vehiculo():

    ruedas=4

    def desplazamiento(self):
        print("El vehículo se está desplazando sobre 4 ruedas")
```

Cuando tenemos nuestra clase lista ya podemos empezar a crear objetos que pertenezcan a esa clase, para crear objetos lo hacemos de la siguiente manera:

```
miCarro = Vehiculo()
```

Después del "=" le estamos especificando a qué clase pertenece el objeto que acabamos de crear.

Para poder mostrar todos los atributos y comportamientos que tiene un objeto a la hora de ejecutar un programa de POO en Python, hacemos lo siguiente:

Para mostrar atributos:

```
miObjeto.atributo
```

Para mostrar métodos:

```
miObjeto.metodo()
```

Siguiendo con el ejemplo, para mostrar en pantalla el atributo y el comportamiento de la clase que le dimos a nuestro objeto "miCarro" lo hacemos de la siguiente manera:

```
print("Mi vehiculo tiene ", miCarro.ruedas, " ruedas")

miCarro.desplazamiento()
```

MÉTODO CONSTRUCTOR

Para crear un objeto de una clase determinada, es decir, instanciar una clase, se usa el nombre de la clase, y a continuación se añaden paréntesis (como si se llamara a una función).

```
nombreObjeto = MiClase()
```

Lo anterior crea una nueva instancia de la clase `MiClase`, y asigna dicho objeto a la variable `nombreObjeto`. Esto crea un objeto vacío, sin estado.

Sin embargo, hay clases (como nuestra clase `Vehículo`) que deben o necesitan crear instancias de objetos con un estado inicial.

Esto se consigue implementando el método especial `__init__()`. Este método es conocido como el constructor de la clase, y se invoca cada vez que se instancia un nuevo objeto.

El método `__init__()` establece un primer parámetro especial que se suele llamar **self**, pero puede especificar otros parámetros siguiendo las mismas reglas que cualquier otra función.

En este caso, el constructor de la clase Vehículo es el siguiente:

```
def __init__(self, color, aceleracion):
    self.color = color
    self.aceleracion = aceleracion
    self.velocidad = 0
```

Como se puede observar, además del parámetro **self**, define los parámetros color y aceleración, que determinan el estado inicial de un objeto de tipo `Vehículo`.

Para instanciar un objeto de tipo coche, debemos pasar como argumentos el color y la aceleración, esto es:

```
carro1 = Vehiculo('Azul', 30)
carro2 = Vehiculo('Balco', 50)
```

MÉTODOS ACCESADORES Y MUTADORES

Existen unos métodos en la interfaz de la clase que puedan permitir al usuario de un objeto acceder y alterar los datos (que se encuentran almacenados internamente en memoria). Por lo tanto, para este caso, tenemos: **Accesadores** (Accessors) y **Mutadores** (Mutators), que son útiles para acceder y alterar, respectivamente, los datos almacenados internamente.

Métodos Accesadores: se utilizan para acceder al estado del objeto, es decir, se puede acceder a los datos ocultos en el objeto desde el método. Sin embargo, éste no puede cambiar el estado del objeto, solo puede acceder a los datos ocultos. Podemos nombrar estos métodos con la palabra **get**.

Métodos Mutadores: se utilizan para mutar o modificar el estado de un objeto, es decir, alterar el valor oculto de la variable de datos. Puede establecer el valor de una variable instantáneamente en un nuevo valor. Este método también se denomina método de actualización. Además, podemos nombrar estos métodos con la palabra **set**.

```
class Vehiculo:
    def __init__(self, color, aceleracion):
        self.color = color
        self.aceleracion = aceleracion
        self.velocidad = 0

    def set_color(self, color):
            self.color = color

    def get_color(self):
            return self.color

miCarro1 = Vehiculo('Negro', '20')
print (miCarro1.get_color())
miCarro1.set_color('Blanco')
print (miCarro1.get_color())
```

¿QUÉ ES LA COLABORACIÓN ENTRE OBJETOS?

Generalmente, en un programa de Programación Orientada a Objetos no participa una sola clase, sino que hay muchas clases que dependen unas de otras para hacer soluciones del problema más pequeñas, y unirlas para obtener el completo, técnica de “divide y vencerás”. Por lo tanto, la colaboración entre objetos es cuando un objeto envía mensajes a otro.

```
class Vehiculo:
    def __init__(self, marca, color, ruedas):
        self.marca = marca
        self.color = color
        self.ruedas = ruedas

class datosVehiculos:
    def __init__(self):
        self.vehiculo = Vehiculo('TOYOTA', 'Rojo', 4)
        print('Datos del vehiculo:')
        print('Marca: ', self.vehiculo.marca)
        print('Color: ', self.vehiculo.color)
```

```
        print('Ruedas: ', self.vehiculo.ruedas)

carro = datosVehiculos()
```

¿QUÉ ES LA COMPOSICIÓN DE OBJETOS?

Es cuando una clase hace referencia a uno o más objetos de otras clases como una variable de instancia. Aquí, al usar el nombre de la clase o al crear el objeto, podemos acceder a los miembros de una clase dentro de otra. Permite crear tipos complejos mediante la combinación de objetos de diferentes clases. Esto es, que una clase compuesta puede contener un objeto de otra clase componente. La composición significa utilizar objetos dentro de otros objetos sin usar herencia.

```
class Componente:
    def __init__(self):
        print('Objeto de la clase componente ha sido creado')
    def metodo_1(self):
        print('Se ejecuta el metodo_1() de la clase Componente')

class Composicion:
    def __init__(self):
        self.objeto_1 = Componente()
        print('Objeto de la clase Composición ha sido creado')
    def metodo_2(self):
        print('Se ejecuta el metodo_2() de la clase Composicion')
        self.objeto_1.metodo_1()

objeto_2 = Composicion()
objeto_2.metodo_2()
```

Salida:

```
Objeto de la clase componente ha sido creado
Objeto de la clase Composición ha sido creado
Se ejecuta el metodo_2() de la clase Composicion
Se ejecuta el metodo_1() de la clase Componente
```

Sobrecarga de Métodos: o comúnmente llamada *Overloading,* es una práctica que consiste en tener diferentes métodos con el mismo nombre en una misma clase, y que el intérprete o compilador logre diferenciarlos por los tipos de datos que se envían como argumentos para los parámetros.

Python no admite la sobrecarga de métodos de forma predeterminada. El problema con la sobrecarga de métodos en Python es que podemos sobrecargar los métodos, pero solo podemos usar el último método definido.

```
def product(a, b):
    p = a * b
    print(p)

def product(a, b, c):
    p = a * b*c
    print(p)


product(4, 5, 5)
```

Salida: 100

En el código anterior, hemos definido dos métodos de producto, pero solo podemos usar el segundo, ya que Python no admite la sobrecarga de métodos. Podemos definir muchos métodos con el mismo nombre y diferentes argumentos, pero solo podemos usar el último método definido. Llamar al otro método producirá un error. Como aquí, la llamada `product(4, 5)` producirá un error, ya que el último método de producto definido toma tres argumentos.

Por lo tanto, para superar el problema anterior, podemos utilizar diferentes formas de lograr la sobrecarga del método.

Método 1 (no es el más eficiente):

Podemos usar los argumentos para hacer que la misma función funcione de manera diferente, es decir, según los argumentos.

```
def add(datatype, *args):
     if datatype =='int':
        answer = 0
     if datatype =='str':
        answer =''
     for x in args:
        answer = answer + x
        print(answer)
        add('int', 5, 6)
```

```
10          add('str', 'Hola')
```

Salida:

```
11
Hola
```

El problema con el código anterior es que hace que el código sea más complejo con múltiples declaraciones if / else, y no es la forma deseada de lograr la sobrecarga del método.

Método 2 (eficiente):

Mediante el uso del decorador de despacho múltiple, este puede ser instalado por:

```
pip3 instalar multipledispatch
from multipledispatch import dispatch
@dispatch(int,int)

def product(first,second):
    result = first*second
    print(result);

@dispatch(int,int,int)
def product(first,second,third):
    result  = first * second * third
    print(result);

@dispatch(float,float,float)
def product(first,second,third):
    result  = first * second * third
    print(result);


product(2,3,2)
product(2.2,3.4,2.3)
```

Salida:

```
12
17.985999999999997
```

DIFERENCIA ENTRE COLABORACIÓN Y COMPOSICIÓN

Como dijimos previamente, la colaboración entre objetos consiste en el proceso en que un objeto envía mensajes a otro para alcanzar la solución deseada.

Por su parte la composición utiliza objetos dentro de otros, es decir, requiere del objeto como parte de si mismo para poder alcanzar la solución deseada.