n2-poo-clases

May 27, 2024

1 Programación Orientada a Objetos

[ ]:
```
alumnos = {
            "111":{"nombre": "Juan","nota":4.5},
            "222":{"nombre": "Lina","nota":3},
            "333":{"nombre": "Pedro","nota":2.8},
          }
#print (alumnos)
cedula = input("Ingrese la cedula del alumno a buscar:")
#print ("El alumno con cedula", cedula, "es ", alumnos[cedula])
print ("El alumno con cedula", cedula, "se llama ", alumnos[cedula]["nombre"],␣
 ↪"y obtuvo una nota de ", alumnos[cedula]["nota"])
```

Es un paradigma de programación en el que los protagonistas son los objetos. Cada objeto es una instancia de una clase y contiene su propia estructura (datos/propiedades) y comportamiento (métodos).

¿Por qué la POO?

¿Ventajas?

1.1 Definiendo las clases

sintaxis básica:

```
class mi_clase:
    miembros (atributos, constructores, métodos)
```

1.1.1 Fundamentos

Modificadores de acceso El guión bajo _ tiene un significado especial en python. En las clases cualquier miembro (atributo, método) por defecto es público. Si el miembro tiene un guión bajo antes de su nombre, significa que es protegido y si tiene dos guiones bajos antes de su nombre, significa que es privado.

```
nombre     #atributo público
_nombre    #atributo protegido
__nombre   #atributo privado

obtener_nombre     #método público
```

```
_obtener_nombre    #método protegido
__obtener_nombre   #método privado
```

[ ]:
```python
class Alumno:
  def __init__(self, nombre, nota):
    self.__nombre = nombre
    self.__nota = nota
    print ("Estoy dentro del constructor")

  def mostrar_info(self, maximo=5):
    print ("nombre", self.__nombre, "nota:", self.__nota, "entre ", maximo)
    print ("Objeto actual ", self)
########################################
a1 = Alumno ("Juan", 3)

a1.mostrar_info()


a2 = Alumno ("Margarita", 3)
a2.mostrar_info(4)
print(a2)
print (a2)
```

Métodos mágicos Cuando un método tiene dos guiones bajos al principio y final de sus nombres se denomina un método mágico. Los métodos mágicos no se pueden llamar directamente desde el código como lo hacemos con los demás métodos, son "mágicos" porque Python los llama automáticamente en determinadas situaciones.

```
__metodo_magico__()   #el doble guión bajo antes y después del nombre del método lo identifica
```

El parámetro self Cada clase da origen a diferentes objetos. El parámetro **self** que se recibe en los constructores o métodos de una clase hace referencia al objeto en particular que está accediendo al comportamiento definido en la clase en un momento determinado.

Todos los métodos en Python deben recibir como primer parámetro el parámetro **self** de esta manera se tiene una referencia al objeto actual que está haciendo uso de los miembros de la clase.

```
obtener_nombre(self):
    return self.nombre      #se devuelve el nombre del objeto actual
```

[ ]:
```python
class Persona:
  def saludar(self, nombre):
    print("hola ", nombre)
    print(self)
###############################################
p = Persona()
```

```
p.saludar("Juan")
p1 = Persona()
p1.saludar("Lina")

print("estoy por fuera y soy :", p)
```

1.1.2 El constructor

Si en Python no especificamos un constructor, de igual manera cada clase tendrá un constructor por defecto que nos permite instanciar objetos de esa clase. Miremos:

[ ]:
```
class Persona:
  pass

juan = Persona()
print (juan)
```

En el código anterior, se creó un objeto de la clase Persona, el objeto se llama **juan** y no hace nada, ya que la clase Persona no tiene código. Pero el objeto existe y tiene memoria asignada para su almacenamiento. Veamos

[ ]:
```
class Persona:
  pass

juan = Persona()
print ("memoria juan:", juan) #muestra la posición de memoria
print ("tipo juan:", type(juan)) #muestra de qué tipo es juan

pedro = Persona()
print ("memoria pedro:", pedro) #muestra la posición de memoria
print ("tipo pedro:", type(pedro)) #muestra de qué tipo es juan
```

___ init___ () Para definir el constructor de una clase en python, usamos el método mágico **init** Miremos:

[ ]:
```
class Persona:
  def __init__(self):
    print ("Me estoy creando ", self)
################################################

juan = Persona()   #creamos objetos de la clase Persona
pedro = Persona()
```

[ ]:
```
class Persona:
  def __init__(self, ced, nom, ed):
    print ("Me estoy creando ", self)
    self.cedula = ced
```

```
    self.nombre = nom
    self.edad = ed
##############################################



obj1 = Persona(123, "Luis", 35)   #creamos objetos de la clase Persona
obj2 = Persona(345, "Lina", 20)
print (obj1.cedula, obj1.nombre, obj1.edad)
print(obj1)
print (obj2.cedula, obj2.nombre, obj2.edad)
print(obj2)
```

En Python no tenemos que declarar explícitamente los atributos de una clase. Ellos van a estar siempre almacenados en el parámetro self. Miremos:

[ ]:
```
class Persona:
  def __init__(self, n, e, pepito):
    self.nombre = n
    self.edad = e
    self.ciudad = pepito

  def mostrar_informacion(self):  #mediante el self el método reconoce al␣
 ↪objeto que está invocando el método
    print(self.nombre, self.edad, self.ciudad)
##################################################

juan = Persona("Juan Villa", 23, "Manizales")
pedro = Persona("Pedro Valencia", 40, "Pereira")

pedro.pasaporte = "AX4743"
pedro.novia = "Florencia"

juan.alergia = "mariscos"

print (juan.ciudad)
print (pedro.ciudad)

print (juan.alergia)

#juan.mostrar_informacion()
#pedro.mostrar_informacion()
```

En Python se pueden crear atributos “al vuelo”, es decir, después de crear un objeto se le pueden asignar los atributos deseados y estos formarán parte de los miembros de este objeto, los cuales recordemos podemos acceder a través del parámetro **self** Veamos:

[ ]:
```
class Persona:
  def mostrar_informacion(self):
    print(vars(self))      # devuelve un diccionario con los atributos y␣
 ↪valores del objeto actual

  def __str__(self):
     return "Hola soy el objeto con esta información: " + str(vars(self))
################################################

juan = Persona()
juan.altura = 180
juan.peso = 85

juan.mostrar_informacion()
print (juan)

obj = Persona()
obj.sexo = "masculino"
obj.salario = 1000000
obj.tipo_sangre = "o+"
obj.mostrar_informacion()

print (obj)
```

1.1.3 Otros métodos mágicos

Las clases tienen por defecto algunos métodos mágicos que podemos sobreescribir para alterar su funcionamiento por defecto.

####___ str___ ()

El método mágico **str** en una clase tiene el comportamiento por defecto de retornar una cadena con la clase a la que pertenece el objeto y su dirección de memoria. Este método mágico es llamado cada vez que a la función **str()** le enviamos por parámetro un objeto. Miremos:

[ ]:
```
class Lenguaje:
  def __init__(self, nombre):
    self.nombre = nombre

  def __str__(self):
    return self.nombre
##########
l1 = Lenguaje("Java")
print(l1)

l2 = Lenguaje("Python")
print(l2)
```

```
l3 = Lenguaje("C#")
print(l3)
```

[ ]:
```
class Persona:
  def __init__(self):
    print (str(self))
##############################################

juan = Persona()   #creamos objetos de la clase Persona
print(juan)
```

En toda clase podemos sobreescribir el método mágico **str** para obtener otro tipo de resultado al imprimir un objeto de dicha clase

[ ]:
```
class Persona:
  def __str__(self):
    return "Este es un objeto cool otra vez"


##############################################

juan = Persona()   #creamos objetos de la clase Persona y nos ejecuta el␣
 ↪constructor init
print (juan)
```

[ ]:
```
class Persona:
  def __init__(self, nombre, edad, direccion):
    self.nombre = nombre
    self.edad = edad
    self.direccion = direccion

  def __str__(self):
    return self.nombre

##############################################

objeto1 = Persona("Andres Julian Valencia", 23, "calle 3")
print (objeto1)
objeto2 = Persona("Ximena Diaz", 35, "calle 66")
print (objeto2)
```

1.1.4 Atributos de Instancia y de Clase

Los atributos de los ejemplos anteriores son atributos de instancia, esto quiere decir que cada instancia (objeto) tiene su propio conjunto de atributos con sus propios valores. Estos atributos son accedidos a través del parámetro **self** y el cambio de valor en un atributo de instancia no tiene efecto en los demás objetos. Incluso, gracias a los atributos dinámicos en Python, dos instancias

de la misma clase pueden tener atributos diferentes con valores por supuesto diferentes. Miremos:

[ ]:
```
class Persona:
  def __init__(self):
    pass
  def mostrar_informacion(self):
    print(vars(self))      # devuelve un diccionario con los atributos y␣
 ↪valores del objeto actual
################################################

juan = Persona()
juan.altura = 180
juan.peso = 85

pedro = Persona()
pedro.altura = 170
pedro.telefono = "300232323"

print ("Información del objeto juan:")
juan.mostrar_informacion()
print ("Información del objeto pedro:")
pedro.mostrar_informacion()
print ("Cambiamos la altura de pedro")
pedro.altura = 200
print ("Información del objeto juan:")
juan.mostrar_informacion()
print ("Información del objeto pedro:")
pedro.mostrar_informacion()
```

Un atributo de clase por su parte, es un atributo cuyo valor es el mismo para todas las instancias de una clase pues pertenece a la clase y no a sus instancias, aunque estas lo puedan acceder. En el siguiente código vamos a usar un atributo de clase para llevar la cuenta del número total de personas creadas:

[ ]:
```
class Persona:
  personas_total = 0 #este es un atributo de clase
  def __init__(self):
    Persona.personas_total += 1  #cada vez que se crea un objeto se incrementa␣
 ↪el atributo de clase
################################################

juan = Persona()
print ("personas_total a través de juan: ", juan.personas_total)


pedro = Persona()
print ("personas_total a través de pedro: ", pedro.personas_total)
print ("personas_total a través de juan: ", juan.personas_total)
```

```
luis = Persona()
print ("personas_total a través de luis: ", luis.personas_total)
print ("personas_total a través de juan: ", juan.personas_total)
print ("personas_total a través de pedro: ", pedro.personas_total)
```

1.1.5 Métodos

Los métodos se definen igual que una función con la diferencia de siempre recibir el primer parámetro **self** que como vimos antes hace referencia al objeto actual.

Sintaxis:

```
def mi_metodo(self):
    cuerpo del método

def mi_metodo(self, parametro1, parametro2, parametro_n):
    cuerpo del método
```

[ ]:
```
class Persona:
  def __init__(self, n, e, c):
    self.nombre = n
    self.edad = e
    self.ciudad = c

  def cumplir_anos(self):
    self.edad += 1

  def get_edad(self):
    return self.edad

  def set_edad(self, x):
    self.edad = x

  def puede_votar(self):
    if self.edad >= 18:
      return True
    else:
      return False

  def cambiar_ciudad(self, nueva_ciudad):
    self.ciudad = nueva_ciudad

  def mostrar_info(self):
    print (f"{self.nombre} tiene {self.edad} años y vive en {self.ciudad}.␣
 ↪Podrá votar??? {self.puede_votar()}")
##########################################################################################
```

```
p = Persona("Raul Diaz", 17, "Manizales")
p.mostrar_info()
p.cumplir_anos()
p.cambiar_ciudad("Cali")
p.mostrar_info()
```

1.1.6 Métodos de Clase

De la misma manera que existen los atributos de clase, también existen los métodos de clase. Un método de clase es un método que tiene acceso a la clase (atributos de clase u otros métodos de clase).

Para declarar un método como método de clase debemos poner el decorador **@classmethod** antes de la declaración del método de clase.

Estos métodos en lugar de recibir como primer parámetro self, reciben como primer parámetro **cls** que representa a la clase y a través de este parámetro puede acceder a los miembros de la clase.

[ ]:
```
class Persona:
  ciudad = "Manizales"  #atributo de clase

  def __init__(self, nombre, edad):
    self.nombre = nombre
    self.edad = edad

  def cumplir_anos(self):
    self.edad += 1


  @classmethod    #este es un método de clase
  def cambiar_ciudad(cls, nueva_ciudad):
    cls.ciudad = nueva_ciudad

  def mostrar_info(self):
    print (f"{self.nombre} tiene {self.edad} años y vive en {Persona.ciudad} ")
###########################################################################

p1 = Persona("Hugo", 35)
p2 = Persona("Paco", 20)
p3 = Persona("Luis", 29)

p1.mostrar_info()
p2.mostrar_info()
p3.mostrar_info()


p1.cambiar_ciudad("Cali")  #la invocación de un método de clase se realiza a␣
 ↪través de la clase
```

```
p1.mostrar_info()
p2.mostrar_info()
p3.mostrar_info()

Persona.ciudad = "Bogotá"
p1.mostrar_info()
p2.mostrar_info()
p3.mostrar_info()
```

1.1.7 Métodos Estáticos

Los métodos estáticos no acceden ni modifican el estado de los objetos (self) ni de las clases (cls). Son métodos que cumplen una función particular y que quedan asociados a una determinada clase.

Para declarar un método como método estático debemos poner el decorador **@staticmethod** antes de la declaración del método estático.

Estos métodos no reciben ni el parámetro self ni el parámetro cls ya que como lo dijimos antes no conocen ni modifican el estado de la clase ni de sus objetos.

[ ]:
```
import random
class Persona:
  def __init__(self, nombre, edad):
    self.nombre = nombre
    self.edad = edad

  @staticmethod    #este es un método estático
  def mensaje_hoy():
    mensajes =["La vida es bella", "Mejor tarde que nunca", "Dios te bendiga",␣
 ↪"Vive para servir"]
    print (random.choice(mensajes))

  def mostrar_info(self):
    print (f"{self.nombre} tiene {self.edad} años")
##########################################################################################
Persona.mensaje_hoy()
p1 = Persona ("Pablo", 38)
p1.mostrar_info()
Persona.mensaje_hoy()
```

1.2 Apropiación

1.

Cree una clase llamada **Estudiante** con los siguientes atributos de instancia: **nombre**, **nota1** y **nota2**.

Defina un constructor que reciba como parámetros el nombre, la nota1 y la nota2 del estudiante.

Defina los siguientes métodos de instancia: **obtener_nota_promedio()** este método devuelve la nota promedio del estudiante y **mostrar_informacion()** este método muestra en pantalla todos los datos del estudiante (nombre, nota1, nota2 y nota promedio).

Pruebe el funcionamiento de la clase.

2.

Encapsule la clase **Estudiante** creada en el punto anterior, de tal manera que todos sus atributos de instancia queden **privados**. Agregue los métodos necesarios dentro de la clase para que continúe su funcionamiento y adicionalmente solo acepte notas entre 0 y 5.

3.

A la clase **Estudiante** del punto anterior realícele las adaptaciones necesarias para que al imprimir un objeto completo de esta clase se presente su nombre y nota promedio.

4.

Requerimos que cualquier estudiante creado pertenezca a la misma institución de tal manera que si un estudiante cambia de institución, todos deben hacerlo. Lleve además un control del número de estudiantes de la institución. Realice los ajustes a la clase **Estudiante** del punto anterior para lograr estos objetivos.

5.

Realice el método ver_escala() que al ser invocado en la clase Estudiante presente una tabla con la escala de calificación de todos los estudiantes de la institución. Esta escala es la siguiente:

Permita que al invocar al método desde la clase **Estudiante.ver_escala()** se muestre la tabla anterior. Utilice el módulo **tabulate** para que la presentación sea la indicada.