



INSTITUTO POLITÉCNICO
NACIONAL



UNIDAD PROFESIONAL INTERDISCIPLINARIA EN
INGENIERÍA Y TECNOLOGÍAS AVANZADAS

PROGRAMACIÓN AVANZADA 2MV7

Practica 3

Autor:

Barrios Mendez Jose Alberto

Boleta: 2022640111

Profesor:

Cruz Mora Jose Luis

Ing. Mecatrónica

13 de marzo de 2024

Índice

1. Objetivo.	3
2. Introduccion.	3
3. Desarrollo.	3
3.1. Computadoras.	3
3.2. Mascotas.	6
4. Resultados	9
4.1. Computadora	9
4.2. Mascotas	9
5. Conclusiones.	12

1. Objetivo.

Desarrollar programas utilizando la herencia y polimorfismo.

2. Introduccion.

Para esta practica continuamos con la programacion orientada a objetos en python, en el desarrollo de esta practica tendremos la implementacion de una tienda de mascotas, la cual esta dividida en categorias, donde aplicaremos la herencia, que es un pilar fundamental de la programacion orientada a objetos. Ademas tendremos una seccion donde desarrollaremos la herencia en clases de computadoras, crearemos objetos distintos como lo son telefonos, pc, laptops.

Estas practicas se desarrollan con el fin de implementar la herencia asi que lo trataremos mas a detalle a continuacion:

3. Desarrollo.

3.1. Computadoras.

Para el desarrollo del codigo debemos percatarnos que vamos a tener una herencia desde una clase de tipo Interfaz hacia 4 clases diferentes, las cuales son: Computadora portatil, Computadora de Escritorio, Telefono inteligente, Tablet. Donde a pesar que estas clases son diferentes todas van a heredar los atributos del clase abstracta. En la clase abstracta vamos a tener los atributos de Memoria, Procesador, Almacenamiento, GPU. ademas de tener declarados sus respectivos getters y setters de forma abstracta, es decir, solo se declara para que el programa este al tanto de los metodos que las clases que heredan de la clase computadora deberan implementarlos al crear una instancia, o por decirlo de otra forma, obligarlos a declarar todos sus atributos cada que se crea un objeto. vamos a analizar el codigo. Primero creamos la clase Computadora, declaramos el iniciador con los atributos antes mencionados, para aplicar el encapsulamiento nosotros declaramos los atributos como privados, aunque en python estos no aplican del todo, es bueno considerarlos para llevar acabo una buena practica de programacion. Cabe recordar que en python para crear los atributos privados es de la forma . __variable

```
class Computadora(metaclass=ABCMeta):
    def __init__(self, memoria, procesador, almacenamiento, gpu):
        self.__memoria=memoria
        self.__procesador=procesador
        self.__almacenamiento=almacenamiento
        self.__gpu=gpu
```

Figura 1: Creacion de la clase Computadora con sus respectivos atributos privados

Posteriormente, considerando que es una clase abstracta, es importante que todos los getters que declaremos dentro de esta clase tambien deben ser abstractos, esto con la finalidad de obligar al usuario a declarar los atributos al iniciar esta clase, ademas de que no se podran crear las instancias directamente de esta clase.

Para crear los getters abstractos soccorremos al decorador `@property` y del decorador `@abstractmethod`. Obsérvese en la figura 2.

```

14         @property
15         @abstractmethod
16     def procesador(self):
17         pass
18 
```

Figura 2: Creacion de la clase Computadora con sus respectivos atributos privados

Para satisfacer los requerimientos solicitados en la practica, se crean todos los getters de cada atributo, de la misma forma como definimos el getter para el atributo procesador, ilustrado en la imagen, recordando nuevamente que estos getters no se implementan en la clase abstracta, es la clase a la que heredamos quien se va a encargar de sobrescribir el metodo para que indique la forma de este metodo.

Posteriormente vamos a crear las subclases que heredaran de la clase Computadora, es importante mencionar que para cada subclase es necesario sobrescribir todos los getters y setters, ya que como estas van a crear instancias u objetos, no pueden estar definidas de la forma abstracta, cada subclase tendra un atributo adicional, ademas de un metodo respectivamente para el atributo, las subclases son.

- Computadora portatil: Para esta subclase fue necesario agregar un atributo protegido llamado tamaño, el cual indica el tamaño de esta misma, los getters y setters quedarian de la forma:

```

class Computadora_Portatil(Computadora):
    def __init__(self, memoria, procesador, almacenamiento, gpu, tamaño):
        super().__init__(memoria, procesador, almacenamiento, gpu)
        self._tamaño=tamaño

    @property
    def tamaño(self):
        return self._tamaño
    @tamaño.setter
    def tamaño(self, nuevo_tamaño):
        self._tamaño=nuevo_tamaño

```

Figura 3: Subclase con el atributo y getters and setters de tamaño

Es importante mencionar que cada subclase sobrescribe cada setter y getter, pero para fines practicos solo mostraremos los que son diferentes entre cada clase

- Computadora Escritorio: Para esta clase solo vamos a definir el atributo y el metodo con sus respectivos getters and setter para el tipo de bocinas que contiene la computadora de escritorio, veamoslo

```

2  class Computadora_Escritorio(Computadora):
3      def __init__(self, memoria, procesador, almacenamiento, gpu, bocinas):
4          super().__init__(memoria, procesador, almacenamiento, gpu)
5          self._bocinas=bocinas
6
7      @property
8      def bocinas(self):
9          return self._bocinas
10     @bocinas.setter
11     def bocinas(self,nueva_bocinas):
12         self._bocinas=nueva_bocinas

```

Figura 4: Subclase con el atributo y getters and setters de bocinas

- Telefono Inteligente: Para esta subclase, lo que va a hacerla diferente va a ser el atributo camara, que describiria en un principio la capacidad de esta

```

115 class TelefonoInteligente(Computadora):
116     def __init__(self, memoria, procesador, almacenamiento, gpu, camara):
117         super().__init__(memoria, procesador, almacenamiento, gpu)
118         self._camara=camara
119
120     @property
121     def camara(self):
122         return self._camara
123     @camara.setter
124     def camara(self,nueva_camara):
125         self._camara=nueva_camara
126

```

Figura 5: Subclase con el atributo y getters and setters de camara

- Tablet: Para esta subclase la diferencia es que tendremos el atributo pantalla

```

158 class Tablet(Computadora):
159     def __init__(self, memoria, procesador, almacenamiento, gpu, pantalla):
160         super().__init__(memoria, procesador, almacenamiento, gpu)
161         self._pantalla=pantalla
162
163     @property
164     def pantalla(self):
165         return self._pantalla
166     @pantalla.setter
167     def pantalla(self,nueva_pantalla):
168         self._pantalla=nueva_pantalla

```

Figura 6: Subclase con el atributo y getters and setters de pantalla

Una vez que hemos definido cada subclase, queda lo mas importante, que es crear los objetos, como sugerencia o instruccion de la practica, nos pide crear al menos 3 objetos por cada subclase, por lo que tendríamos un total de 12 objetos, como estamos utilizando una interfaz o clase abstracta, es necesario que al crea cada objeto, inicializemos todos sus atributos, con

la ayuda de los setters, podemos modificar cada atributo posterior a su creacion. veamos en la imagen la creacion de los objetos

```

202
203 lap1=Computadora_Portatil(memoria=32,procesador="i3 10th",almacenamiento=512,gpu="Nvidia",tamaño="12 pulgadas")
204 lap2=Computadora_Portatil(memoria=16,procesador="i7 11th",almacenamiento=1024,gpu="Nvidia 3050",tamaño="15 pulgadas")
205 lap3=Computadora_Portatil(memoria=64,procesador="ryzen 7560",almacenamiento=256,gpu="Nvidia 6018",tamaño="14 pulgadas")
206 pc1=Computadora_Escritorio(memoria=16,procesador="i9 11th",almacenamiento=1024,gpu="Nvidia 4080",bocinas="bosse")
207 pc2=Computadora_Escritorio(memoria=32,procesador="Pentium",almacenamiento=256,gpu="none",bocinas="akg")
208 pc3=Computadora_Escritorio(memoria=16,procesador="i9 11th",almacenamiento=1024,gpu="Nvidia 4080",bocinas="bosse")
209 cel1=TelefonoInteligente(memoria=6,procesador="snapdragon 865",almacenamiento=64,gpu="sin datos",camara=24)
210 cel2=TelefonoInteligente(memoria=8,procesador="snapdragon gen 1",almacenamiento=512,gpu="grafic intel",camara=108)
211 cel3=TelefonoInteligente(memoria=4,procesador="A bionic 14",almacenamiento=64,gpu="sin datos",camara=12)
212 tab1=Tablet(memoria=6,procesador="A bionic 16",almacenamiento=512,gpu="apple graphics",pantalla=10.2)
213 tab2=Tablet(memoria=12,procesador="Kirin 980",almacenamiento=512,gpu="huawei graphics",pantalla=9)
214 tab1=Tablet(memoria=8,procesador="snapdragon 780",almacenamiento=256,gpu="sin datos",pantalla=8)
215

```

Figura 7: Creacion de los objetos iniciando sus atributos

3.2. Mascotas.

El objetivo de esta practica es llevar a cabo la herencia en python, la practica nos pide crear varias clases abstractas, comenzando con la clase Mascota, la cual contiene los atributos de: nombre,edad,dueño,tipo. Asi como sus respectivos getters, es importante mencionar que el diagrama de clases nos pide adicionalmente 2 metodos aparte, los cuales son el metodo habla y el metodo toString() que para fines practicos e definido como un metodo str, este metodo se usara para imprimir los datos mas adelante los metodos adicionales son creados de la siguiente forma

```

30     @property
31     @abstractmethod
32     def habla(self):
33         pass
34
35     @property
36     @abstractmethod
37     def __str__(self) -> str:
38         pass
39
40     @__str__.setter
41     @abstractmethod
42     def __str__(self,nuevo_ha) -> str:
43         pass

```

Figura 8: Creacion de los metodos abstractos habla y str

Posteriormente esta clase abstracta sera la plantilla para crear otras 2 clases abstractas, que seran las clases Domestica y Exotica.

Domestica: Para esta clase se nos pide agregar un nuevo atributo que es el de nivel de ternura, con sus respectivo set y get, ya que este valor si puede ser modificado.

```
class Domestica(Mascota):
    def __init__(self, nombre, edad, dueño, tipo, ternura) -> None:
        super().__init__(nombre, edad, dueño, tipo)
        self.__ternura=ternura

    @property
    @abstractmethod
    def factor_ternura(self):
        pass

    @factor_ternura.setter
    @abstractmethod
    def factor_ternura(self, nuevo_factor_ternura):
        pass
```

Figura 9: Creacion de la clase abstracta Domestica con el metodo ternura

Por su contraparte tenemos la clase abstracta, a comparacion con la clase Domestica este va a tener el metodo Peligro.

```
60 = class Exotica(Mascota):
61 =     def __init__(self, nombre, edad, dueño, tipo, peligro) -> None:
62 =         super().__init__(nombre, edad, dueño, tipo)
63 =         self.__peligro=peligro
64 =
65 =     @property
66 =     @abstractmethod
67 =
68 =     def factor_peligro(self):
69 =         pass
70 =     @factor_peligro.setter
71 =     @abstractmethod
72 =     def factor_peligro(self, nuevo_factor_peligro):
73 =         pass
74 =
```

Figura 10: Creacion de la clase abstracta Exotica con el metodo peligro

De la clase Domestica se generan 2 subclases que son Perro y Gato, para estas clases se deben reescribir los metodos, ya que a partir de esta clase si se generan objetos. Para ejemplificar mostraremos los metodos habla y str de la clase Gato

```

105     def __str__(self) -> str:
106         return f"Gato: {self._Mascota_nombre}, Edad: {self._Mascota_edad}, Dueño: {self._Mascota_dueño}, ...
107         |tipo: {self._Mascota_tipo}, Ternura: {self._Domestica_ternura}"
108
109     @property
110     def habla(self):
111         return 'MIAU MIAU'

```

Figura 11: sobre escritura de los metodos habla y str de la clase Gato

Para la herencia de la clase Exotica vamos a encontrar que este genera clases con el atributo peligro, las subclases que heredan esto son: vivora,tigre,dinosaurios. Veamos un ejemplo de la sobre escritura de los metodos habla y str de un dinosaurio

```

255     @property
256     def habla(self):
257         return 'Hola soy un brontosaurio grrrrr'
258
259     def __str__(self) -> str:
260         return f"{self.habla}. Nombre: {self._Mascota_nombre}, Edad: {self._Mascota_edad},
261         Dueño: {self._Mascota_dueño}, Tipo: {self._Mascota_tipo}, Peligro: {self._Exotica_peligro}"
262

```

Figura 12: sobre escritura de los metodos habla y str de la clase Brontosaurio

Una vez que hemos creado todas las clases con ayuda de la herencia, dejamos este archivo para exportarlo como un modulo posteriormente, en un nuevo programa creamos la estructura para el menu dentro al momento de ejecutarlo y este sea de una forma mas legible y con ayuda de funciones para poder validar los datos de entrada, nos es posible crear los objetos, aunado a un ciclo while donde se encuentra la parte principal del codigo:

```

111 while True:
112     print(imprimir_menu())
113     opcion = obtener_opcion("\nIngrese una opcion: ", 3)
114     os.system("cls")
115
116     if opcion == 1:...
139
140
141     elif opcion == 2:
142         print('-'*60)
143         print("MASCOTAS ADQUIRIDAS")
144         print('-'*60)
145         for i in range(len(lista)):
146             print(lista[i])
147         input("Presione cualquier tecla para regresar al menú principal.")
148         os.system("cls")
149
150     elif opcion == 3:
151         print("Hasta la proxima!!! ")
152         break

```

Figura 13: codigo para añadir mascotas como desee el usuario

en el código se puede observar que llamamos la función `menu` para mostrarlo en pantalla, preguntamos sobre la opción que el usuario desea hacer, en caso de ser 1, se despliega otro menú para mostrar las mascotas, en el caso 2, se muestran las mascotas adquiridas, en la opción 3 es para salir del programa

4. Resultados

4.1. Computadora

Al momento de ejecutar el código nos va a mostrar todos los objetos con sus características, tal como se muestra en la siguiente figura.

```
Objetos creados:
Computadoras portátiles:
Laptop 1: memoria=32, procesador=i3 10th, almacenamiento=512, gpu=Nvidia, tamaño=12 pulgadas
Laptop 2: memoria=16, procesador=i7 11th, almacenamiento=1024, gpu=Nvidia 3050, tamaño=15 pulgadas
Laptop 3: memoria=64, procesador=ryzen 7560, almacenamiento=256, gpu=Nvidia 6018, tamaño=14 pulgadas

Computadoras de escritorio:
PC 1: memoria=16, procesador=i9 11th, almacenamiento=1024, gpu=Nvidia 4080, bocinas=bose
PC 2: memoria=32, procesador=Pentium, almacenamiento=256, gpu=none, bocinas=akg
PC 3: memoria=16, procesador=i9 11th, almacenamiento=1024, gpu=Nvidia 4080, bocinas=bose

Teléfonos inteligentes:
Celular 1: memoria=6, procesador=snapdragon 865, almacenamiento=64, gpu=sin datos, camara=24
Celular 2: memoria=8, procesador=snapdragon gen 1, almacenamiento=512, gpu=grafic intel, camara=108
Celular 3: memoria=4, procesador=A bionic 14, almacenamiento=64, gpu=sin datos, camara=12

Tabletas:
Tablet 1: memoria=6, procesador=A bionic 16, almacenamiento=512, gpu=apple grafics, pantalla=10.2
Tablet 2: memoria=12, procesador=Kirin 980, almacenamiento=512, gpu=huawei graphics, pantalla=9
Tablet 3: memoria=8, procesador=snapdragon 780, almacenamiento=256, gpu=sin datos, pantalla=8
PS E:\Programacion\Programacion Avanzada_2MV7\Practica3\Barrios_Mendez_JoseAlberto_Codigos>
```

Figura 14: Ejecución del programa número 1

Es importante mencionar que al usar los setters es posible modificar los valores de cada atributo, esto se puede llevar a cabo de la siguiente forma **`lap1.memoria=67`**, lo cual lo convierte en una forma más sencilla de poder sobrescribir lo que en un principio se definió al momento de crear el objeto

4.2. Mascotas

Al ejecutar el programa nos arroja lo siguiente

```

PS E:\Programacion\Programacion_Avanzada_2MV7\Pract
gramacion/Programacion_Avanzada_2MV7/Practica3/Barr

¡Bienvenido al sistema de adopción de mascotas!
+-----+-----+
| Opción | Descripción |
+-----+-----+
| 1      | Adquirir mascota |
| 2      | Visualizar mascotas |
| 3      | Salir          |
+-----+-----+

Ingrese una opcion: █

```

Figura 15: Resultado al ejecutar el programa

vamos a empezar con añadir una mascota, que es la opcion 1:

```

PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS  CODE REFERENCE LOG

A continuación, las mascotas disponibles:
+-----+-----+-----+
| Tipo de mascota | Número | Especies disponibles |
+-----+-----+-----+
| Domésticas      | 1      | Perro                |
|                  | 2      | Gato                 |
| Exóticas        | 3      | Vívora               |
|                  | 4      | Tigre                |
|                  | 5      | Dinosaurio (Brontosaurio) |
|                  | 6      | Dinosaurio (Raptor)  |
|                  | 7      | Dinosaurio (TRex)   |
+-----+-----+-----+

Seleccione la mascota que desea adoptar: █

```

Figura 16: Segundo menu para elegir una mascota

Al seleccionar una opcion por ejemplo Gato (opcion 4) el programa procede a solicitar los datos:

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL

Ingrese el nombre: asd
Ingrese la edad: 12
Ingrese el nombre del dueño: Alberto
Ingrese el nivel de ternura (0 a 10): 9
Ingrese el tipo: Grande
```

Figura 17: Ingreso de los datos de la mascota seleccionada

Al terminar de introducir los datos, nos arroja lo siguiente:

```
Mascota agregada con exito
+-----+-----+
| Opción | Descripción |
+-----+-----+
| 1      | Adquirir mascota |
| 2      | Visualizar mascotas |
| 3      | Salir          |
+-----+-----+

Ingrese una opcion: 
```

Figura 18: Mascota añadida y vuelta al menu

Para mostrar las mascotas adquiridas es suficiente con seleccionar la opcion numero 2:

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS  CODE REFERENCE LOG

-----
MASCOTAS ADQUIRIDAS
-----
Gato: asd, Edad: 12, Dueño: Alberto, Tipo: Grande, Ternura: 9
Presione cualquier tecla para regresar al menú principal.
```

Figura 19: Visualizacion de las mascotas adquiridas

5. Conclusiones.

Al realizar esta practica, de forma independiente a lo extensa que resulto, aunque la programación orientada a objetos puede no siempre conducir a una reducción directa en la cantidad de líneas de código escritas, sus beneficios en términos de mantenimiento, reutilización, flexibilidad y comprensión del código son invaluableles. Al adoptar los principios de la programación orientada a objetos y aplicarlos de manera efectiva en nuestros proyectos, podemos construir sistemas más robustos, flexibles y fáciles de mantener, lo que finalmente conduce a un desarrollo de software más eficiente y exitoso.