****

**TEMA: ENCAPSULAMIENTO**

**NOMBRE DEL ALUMNO:** Woolfolk Cerecer Brian

**SEMESTRE:** 8vo

**NOMBRE DE LA MATERIA:** Seminario de programación

**CLAVE DE LA MATERIA**: COM35C1

**INTRODUCCIÓN**

En el presente reporte de código, se mostrará un ejemplo de encapsulamiento dentro del lenguaje de programación de Python.

Encapsulamiento es un término utilizado y muy presente de la programación orientada a objetos, es así como esto permite que una propiedad, atributo o método sea de carácter privado o público. Cuando hablamos de “privado” o “público” nos referimos al ámbito desde dónde se accede a dichas propiedades y elementos, por ejemplo, ya sea que accedamos desde el flujo principal de código, desde dentro de una función personalizada, o que únicamente el mismo objeto sea quien manipule sus propiedades a través del elemento “self” (que es el equivalente de Python para el “this” de otros lenguajes de programación).

El encapsulamiento es una práctica bastante robusta que nos ayuda a preservar la seguridad del código y los valores. Ya que la única manera de acceder a una propiedad privada será desde el mismo objeto, lo que da cabida a la posibilidad de crear funciones de verificación de datos e integridad del tipado.

Esta práctica consistirá, entonces, en la creación de una clase protagónica que modele la información básica de una persona. Contendrá propiedades tales como el nombre, la edad y el género. Todas sus propiedades serán privadas, por lo que será necesario destinar métodos de clase para poder acceder a ellas, así como también para editar su información.

Con fines de optimización y ejemplificar el alcance de los métodos *get* y *set*, se dispondrá de una lista global que guarde los géneros válidos disponibles, siendo posible entonces que la propiedad de género guarde el *índice* del valor elegido. Esto nos brinda la oportunidad de crear el método get\_genero que no regrese el valor numérico del índice de la lista, sino que busque dentro de dicha lista y regrese el valor que encuentre en ella (es decir, algo similar a generos[self.\_\_genero]).

Además, esto también implica la verificación de errores que si bien, puede ser leve, es importante para formar mejores proyectos a futuro. Dichas verificaciones son básicas como que el nombre no puede estar vacío, la edad debe ser entero mayor a 0, y el género debe elegirse de la lista anterior.

A fin de reutilizar código, se utilizarán las funciones leer\_string y leer\_entero así como las demás funciones relevantes trabajadas en la actividad anterior número 5 de *Clases y Objetos*.

**DESARROLLO**

|  |
| --- |
| **Código** |
| # -------------------------------- CLASE PERSONA  # QUE GUARDE INFORMACION DE SU NOMBRE, EDAD Y GENERO  class Persona:  # -------------------------------- METODO INICIALIZADOR  # MARCAMOS LOS PARAMETROS COMO OPCIONALES PARA CONSIDERAR DEFAULT  def \_\_init\_\_(  self,  nombre: str | None = None,  edad: int | None = None,  genero: int | None = None  ):  # DECLARAMOS VALORES POR DEFECTO DE LA PERSONA,  # ESTO GARANTIZA LA INTEGRIDAD DE LA INFORMACION  self.\_\_nombre = "Nueva persona"  self.\_\_edad = 1  self.\_\_genero = 2 # REPRESENTA AL GENERO 'OTRO'  # EN CASO QUE EXISTAN PARAMETROS, LOS USAREMOS  # RECORDEMOS QUE LAS FUNCIONES MANEJAN LOS ERRORES  if nombre is not None:  self.set\_nombre(nombre)  if edad is not None:  self.set\_edad(edad)  if genero is not None:  self.set\_genero(genero)  # -------------------------------- METODO PARA MOSTRAR COMO CADENA  def \_\_str\_\_(self):  return (f"Nombre: {self.get\_nombre()}"  f"\nEdad: {self.get\_edad()}"  f"\nGenero: {self.get\_genero()}")  # -------------------------------- FUNCIONES GET  # SIMPLEMENTE PARA REGRESAR EL VALOR PROTEGIDO RESPECTIVAMENTE  def get\_nombre(self):  return self.\_\_nombre  def get\_edad(self):  return self.\_\_edad  def get\_genero(self):  try:  # EN CASO DE QUE SE MOFIQUEN LOS GENEROS, COMPROBAMOS  return generos[self.\_\_genero]  except IndexError:  self.\_\_genero = 0 # CORREGIMOS  return "Error!" # INFORMAMOS DE UN ERROR  # -------------------------------- FUNCIONES SET  # AYUDAN A VERIFICAR LOS TIPOS DE DATOS Y RESTRICCIONES  def set\_nombre(self, valor: str):  if string\_valido(valor) is not None:  self.\_\_nombre = valor  def set\_edad(self, valor: int):  if entero\_valido(valor, True) is not None:  self.\_\_edad = valor  def set\_genero(self, valor: int):  if indice\_valido(valor, len(generos)) is not None:  self.\_\_genero = valor  # -------------------------------- FUNCIONES DE COMPROBACION DE TIPOS  # REVISA SI EL STRING ES VALIDO Y LO REGRESA, SINO REGRESA None  def string\_valido(valor) -> str | None:  if not isinstance(valor, str) or valor == "":  print("--- Ingrese una cadena de texto valida!\n")  return None  return valor # ESTE VALOR ES VALIDO  # REVISA SI EL ENTERO ES VALIDO Y LO REGRESA, SINO REGRESA None,  # TAMBIEN NOS PERMITE INDICAR SI QUEREMOS QUE SEA POSITIVO  def entero\_valido(valor, es\_positivo: bool) -> int | None:  if not isinstance(valor, int) or (es\_positivo and valor <= 0):  if es\_positivo:  print("--- Ingrese un numero entero mayor a 0!\n")  else:  print("--- Ingrese un numero entero!\n")  return None  return valor # ESTE VALOR ES VALIDO  # REVISA SI UN NUMERO REPRESENTA UN INDEX DE UNA LISTA  # EL PARAMETRO tamanio\_maximo REPRESENTA EL len(lista)  def indice\_valido(valor, tamanio\_maximo: int) -> int | None:  # EN CASO DE QUE LA LISTA ESTE VACIA, REGRESAMOS -1  if tamanio\_maximo == 0:  print("--- Esta vacio!\n")  return -1 # GRACIAS AL RESTO DE VERIFICACIONES, ESTO PREVIENE ERRORES  # VERIFICAMOS SI EL NUMERO ES VALIDO PRIMERO,  # SUMAMOS 1 EN CASO DE QUE SEA 0, PARA TOMARLO COMO VALIDO  if entero\_valido(valor + 1, True) is None:  return None # SIGNIFICA QUE EL NUMERO NO ERA VALIDO  # DESPUES, VERIFICAMOS SI EL NUMERO ENTRA DENTRO DEL RANGO VALIDO,  if not (tamanio\_maximo > valor >= 0):  print("--- Ingrese un numero valido de la lista!\n")  return None # NO EXISTE EN LA LISTA  return valor # ESTE VALOR ES VALIDO  # -------------------------------- FUNCIONES DE LECTURA RAPIDA  # FUNCION PARA OBTENER UNA CADENA DE TEXTO VALIDA 100%  def leer\_string(mensaje: str) -> str:  # CREAMOS UN CICLO 'INFINITO' PARA INSISTIR CON LA CAPTURA  while True:  try:  # SOLICITAMOS EL VALOR Y LO COMPROBAMOS  valor = string\_valido(input(mensaje))  # SI LA COMPROBACION FUNCIONA, REGRESAMOS EL VALOR  if valor is not None:  return valor # CIERRA EL CICLO CON UN VALOR CORRECTO  # NO ES NECESARIO MOSTRAR ERRORES GRACIAS A string\_valido  # continue  except ValueError:  # 'ATRAPAMOS' CUALQUIER ERROR INESPERADO Y REINICIAMOS EL CICLO  print("--- Ingrese una cadena de texto valida!\n")  # FUNCION PARA OBTENER UN NUMERO ENTERO VALIDO 100%,  # PUDIENDO DECIR SI QUEREMOS QUE SEA POSITIVO O NO  def leer\_entero(mensaje: str, es\_positivo: bool) -> int:  # CREAMOS UN CICLO 'INFINITO' PARA INSISTIR CON LA CAPTURA  while True:  try:  # SOLICITAMOS EL VALOR, LO CONVERTIMOS A int Y LO COMPROBAMOS  valor = entero\_valido(int(input(mensaje)), es\_positivo)  # SI LA COMPROBACION FUNCIONA, REGRESAMOS EL VALOR  if valor is not None:  return valor # CIERRA EL CICLO CON UN VALOR CORRECTO  # NO ES NECESARIO MOSTRAR ERRORES GRACIAS A entero\_valido  # continue  except ValueError:  # 'ATRAPAMOS' CUALQUIER ERROR INESPERADO Y REINICIAMOS EL CICLO  print("--- Ingrese un numero valido!\n")  # FUNCION PARA ESCOGER ENTRE UNA LISTA DE OPCIONES (Y TAMBIEN MUESTRA LAS OPCIONES)  def seleccionar\_indice(mensaje: str, lista: list) -> int:  # EN CASO DE QUE LA LISTA ESTE VACIA, REGRESAMOS -1  if len(lista) == 0:  print("--- Esta vacio!\n")  return -1 # GRACIAS AL RESTO DE VERIFICACIONES, ESTO PREVIENE ERRORES  # CREAMOS UN CICLO 'INFINITO' PARA INSISTIR CON LA CAPTURA  while True:  try:  # PRIMERO MOSTRAMOS LAS OPCIONES  print("Escriba el numero de la opcion que desee:")  mostrar\_lista(lista) # USAMOS mostrar\_lista  # DESPUES SOLICITAMOS EL VALOR Y LO VERIFICAMOS  # COMO LA LISTA COMIENZA EN 1, RESTAMOS PARA INCLUIR EL INDICE 0  valor = indice\_valido(int(input(mensaje)) - 1, len(lista))  # SI LA COMPROBACION FUNCIONA, REGRESAMOS EL VALOR  if valor is not None:  return valor # CIERRA EL CICLO CON UN VALOR CORRECTO  # NO ES NECESARIO MOSTRAR ERRORES GRACIAS A indice\_valido  # continue  except ValueError:  # 'ATRAPAMOS' CUALQUIER ERROR INESPERADO Y REINICIAMOS EL CICLO  print("--- Ingrese un numero valido de la lista!\n")  # FUNCION PARA MOSTRAR UNA LISTA JUNTO A SUS INDICES  def mostrar\_lista(lista: list):  indice = 0 # CREAMOS UN CONTADOR  for item in lista:  # MOSTRAMOS EL INDICE Y EL VALOR  # EL INDICE SUMA 1 PARA CONTAR CON NUMEROS NATURALES  print(f" {indice + 1}. {item}")  indice += 1 # AVANZAMOS  # SI EL INDICE NUNCA CAMBIA, ES QUE NO HAY ITEMS EN LA LISTA  if indice == 0:  print(" Esta vacio!")  print("") # SALTO DE LINEA  # -------------------------------- COMENZAR EL PROGRAMA  print("=== ACTIVIDAD 6 - ENCAPSULAMIENTO ===\n")  # INICIALIZAMOS LA LISTA DE GENEROS VALIDOS COMO GLOBAL  generos = ["Masculino", "Femenino", "Otro"]  # CREAR A persona1 JUAN, 30 Y MASCULINO  persona1 = Persona("JUAN", 30, 0)  # IMPRIMIMOS LA PERSONA POR DEFECTO  # RECORDEMOS QUE EL METODO \_\_str\_\_ UTILIZA GET  print(persona1)  print("") # SALTO DE LINEA  # MODIFICAR A persona1 PARA MARIA, 25 Y FEMENINO  # SOLICITAMOS AL USUARIO PARA MODIFICAR LOS DATOS  persona1.set\_nombre(leer\_string("Ingrese un nuevo nombre: "))  persona1.set\_edad(leer\_entero("Ingrese una nueva edad: ", True))  # UTILIZANDO LA FUNCION seleccionar\_indice PODEMOS ESCOGER EL GENERO FACILMENTE  persona1.set\_genero(seleccionar\_indice("Ingrese un nuevo genero: ", generos))  # IMPRIMIMOS NUEVAMENTE A LA persona1 PARA MOSTRAR LOS CAMBIOS  print("\nDatos modificados:\n")  print(persona1) |

|  |
| --- |
| **Visualización de los resultados del código** |
| Imagen 1. Inicio del programa, mostrando los datos previos de JUAN. Al mismo tiempo que se captura la nueva información.    Imagen 2. Solicitar al usuario el genero nuevo, utilizando la función *seleccionar\_indice* que además cuenta con sus respectivas verificaciones.    Imagen 3. Finalmente se muestra la información modificada, donde apreciamos que el género se accede gracias al índice anteriormente capturado. |

**CONCLUSIÓN**

La aplicación es sencilla, puesto que únicamente nos permite cambiar la información de la persona a una nueva; sin embargo, su verdadera funcionalidad es la de demostrar claramente cuáles son las ventajas de utilizar métodos de lectura y escritura dedicados.

Las propiedades se mantuvieron privadas en todo momento, y las funciones de lectura (especialmente get\_genero con su función de buscar el índice dentro de la lista generos) nos ayudaron a visualizar la información de mejor manera (utilizando la función \_\_str\_\_ para obtener una representación en cadena de texto).

Las funciones creadas previamente en la Actividad 5 de Clases y Objetos también jugaron un rol fundamental en el sistema, ya que fueron las principales medidas de verificaciones y manejo de errores. Estas funciones también representan la esencia de las librerías de código, ya que se trata de métodos comunes con múltiples funcionalidades que se adaptan a cualquier nuevo problema y sistema. Bajo la buena práctica de DRY (*Don’t Repeat Yourself*) podemos empezar a crear una librería de funciones “estándar” y multipropósito para irlas mejorando con forme el paso de actividades, y según la experiencia que acumulemos en el proceso.

Al tratarse de un ejercicio didáctico, no existen muchas mejoras o modificaciones que valga la pena realizar. Es sencillo crear un menú de opciones para modificar un atributo en específico, o para crear y almacenar una lista de personas. Sin embargo, estas funciones van después del propósito de la actividad que es demostrar el uso y ventajas del encapsulamiento, por lo que no se consideraron pertinentes ni necesarias de realizarse.

**REFERENCIAS BIBLIOGRÁFICAS**

Python Software Foundation. (s.f.). *Classes*. Python documentation. <https://docs.python.org/3/tutorial/classes.html>