

# **Documentación y guía paso a paso de** **Sistema ABM Librería**

## Tabla de contenido

1. Introducción
  - 1.1. Alcance
  - 1.2. Comportamiento
  - 1.3. Maqueta (Previsualización)
  
2. Reproducción paso a paso del sistema
  - 2.1. Introducción de herramientas a utilizar
  - 2.2. Configuración del área de trabajo
  - 2.3. Creación de la estructura MVC
  - 2.4. Desarrollo de apartado visual (vista.py)
    - 2.4.1. Sector Principal
    - 2.4.2. Listado de registros
    - 2.4.3. Sector de búsqueda/filtro
  - 2.5. Lógica del sistema (modelo.py)
  - 2.6. Uniendo el apartado visual y la logica (main.py)

# **1. Introduccion**

## **1.1 Alcance**

La finalidad de la aplicación a desarrollar es la de mantener un control del catálogo de los libros e historietas presentes dentro de una librería.

Para esto se pensó una aplicación en donde se ingresen los siguientes datos:

- Título: Título comercial del libro
- Precio: Valor en pesos del producto
- Editorial: La empresa que produjo ese libro
- Genero: Genero literario del libro
- Autor: Escritor del libro

## **1.2 Comportamiento**

La aplicación se compone de seis campos de ingreso de texto junto a seis etiquetas que se encargan de describir cada uno de estos campos, a su vez, la aplicación cuenta con un listado en

donde se exhiben todos los registros guardados y una serie de cinco botones que se comportan según se describe a continuación:

- Guardar: Almacena los datos ingresados en los campos de entrada dentro de la Base de Datos.
- Actualizar: Edita y guarda un registro seleccionado desde la lista de libros.
- Eliminar: Elimina desde la Base de Datos al registro seleccionado en el listado.
- Buscar: Muestra un listado de registros que tengan un título parecido o igual al ingresado dentro del campo de texto “Buscar por título”.
- Borrar filtro: Muestra el listado completo de registros y limpia el campo de texto “Buscar por título”

## 1.3 Maqueta (Previsualización)

Proyecto Analisis de Sistema/POO

### Sistema ABM Librería

Título	<input type="text"/>
Precio	\$ <input type="text"/>
Editorial	<input type="text"/>
Genero	<input type="text"/>
Autor	<input type="text"/>

ID	Título	Precio	Editorial	Genero	Autor	Ult. Actualizacion
----	--------	--------	-----------	--------	-------	--------------------

**● Etiqueta / Label**

**● Campo de texto / Entry**

**● Listado de registros**

**● Botón**

## **2. Reproducción paso a paso del Sistema**

### **2.1 Introducción de herramientas a utilizar**

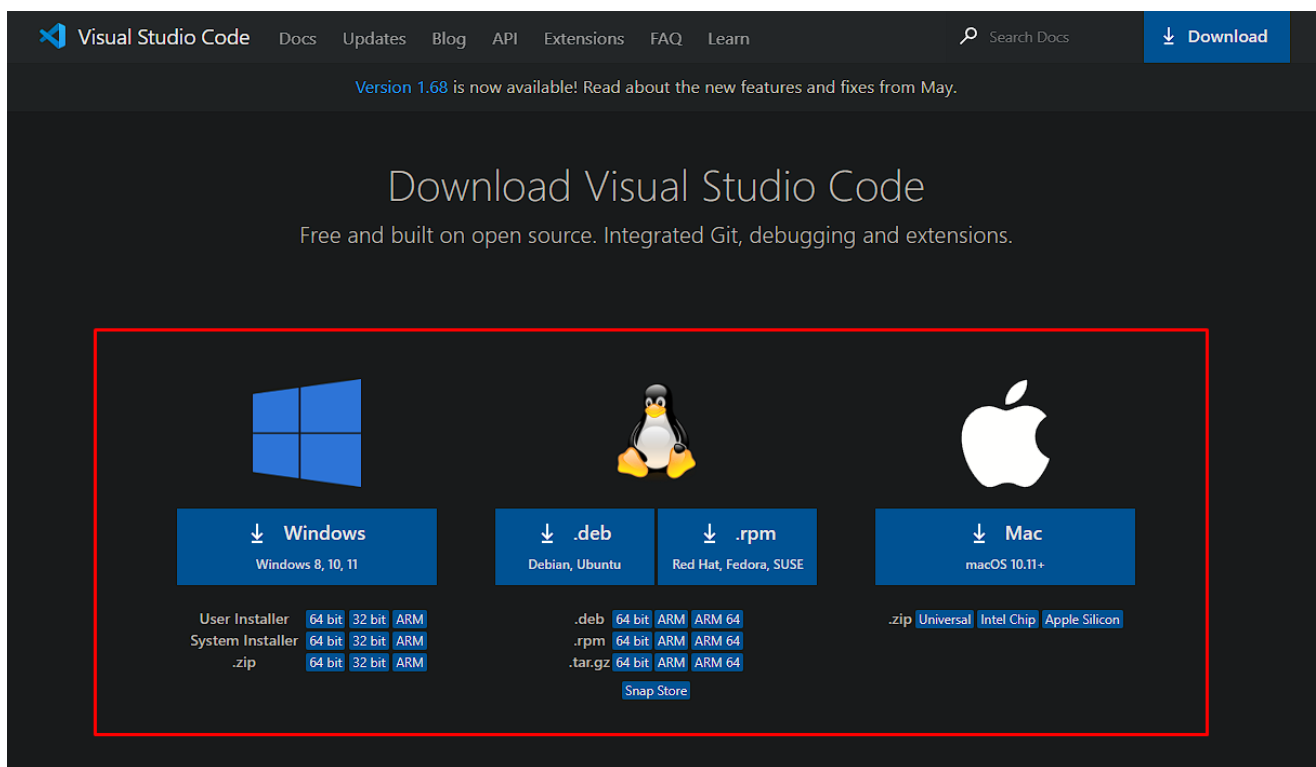
Esta es una breve introducción a las herramientas que utilizo a lo largo del desarrollo de este sistema:

- Visual Studio Code: editor de texto en donde voy a programar el sistema utilizando el lenguaje de programación Python.
- Dentro de Python utilizo las siguientes librerías:
  - Tkinter: interfaz gráfica por la cual el usuario se va a poder comunicar con el sistema
  - SQLite3: motor de bases de datos SQL ligero donde almacenaremos los datos ingresados
  - re: módulo que proporciona operaciones de coincidencia de expresiones regulares (utilizado en las validaciones de ingreso de datos)

- Datetime: módulo que proporciona clases para manipular fechas y horas


## 2.2 Configuración del área de trabajo

El primer paso al momento de configurar el área de trabajos es el de descargar Visual Studio Code, para eso hay que dirigirse a la pagina <https://code.visualstudio.com/download> en donde hay que hacer click en el enlace de descarga correspondiente a nuestro sistema operativo:



El siguiente paso es el de descargar Python por lo que me voy a dirigir ir al siguiente enlace: <https://www.python.org/downloads/> y hacer click en el botón de descarga de la versión más reciente

[Python](#)[PSF](#)[Docs](#)[PyPI](#)[Jobs](#)[Community](#)



[Donate](#)[GO](#)[Socialize](#)

[About](#)[Downloads](#)[Documentation](#)[Community](#)[Success Stories](#)[News](#)[Events](#)


### Download the latest version for Windows

Download Python 3.10.5

Looking for Python with a different OS? Python for [Windows](#), [Linux/UNIX](#), [macOS](#), [Other](#)

Want to help test development versions of Python? [Prereleases](#), [Docker images](#)

Looking for Python 2.7? See below for specific releases

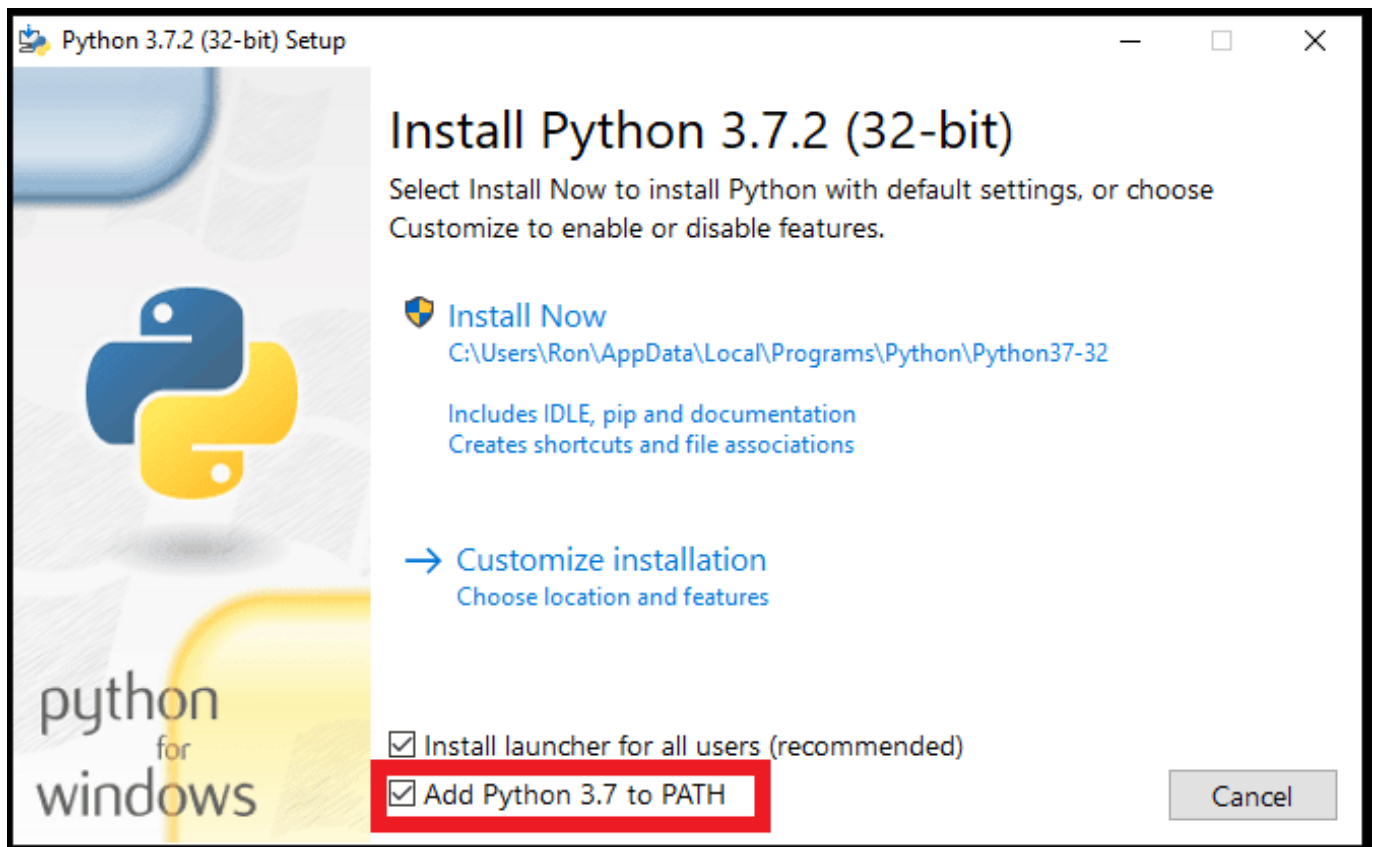


#### Active Python Releases

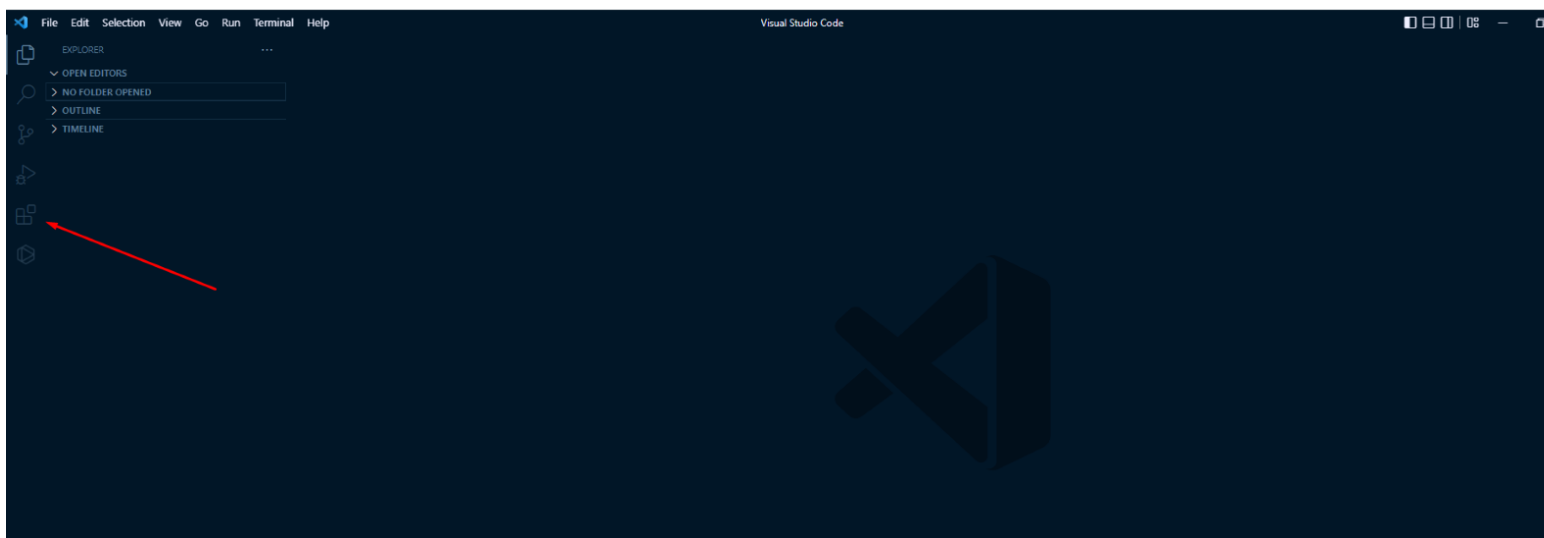
For more information visit the [Python Developer's Guide](#).

Python version	Maintenance status	First released	End of support	Release schedule
3.10	bugfix	2021-10-04	2026-10	<a href="#">PEP 619</a>
3.9	security	2020-10-05	2025-10	<a href="#">PEP 596</a>
3.8	security	2019-10-14	2024-10	<a href="#">PEP 569</a>
3.7	security	2018-06-27	2023-06-27	<a href="#">PEP 537</a>
2.7	end-of-life	2010-07-03	2020-01-01	<a href="#">PEP 373</a>

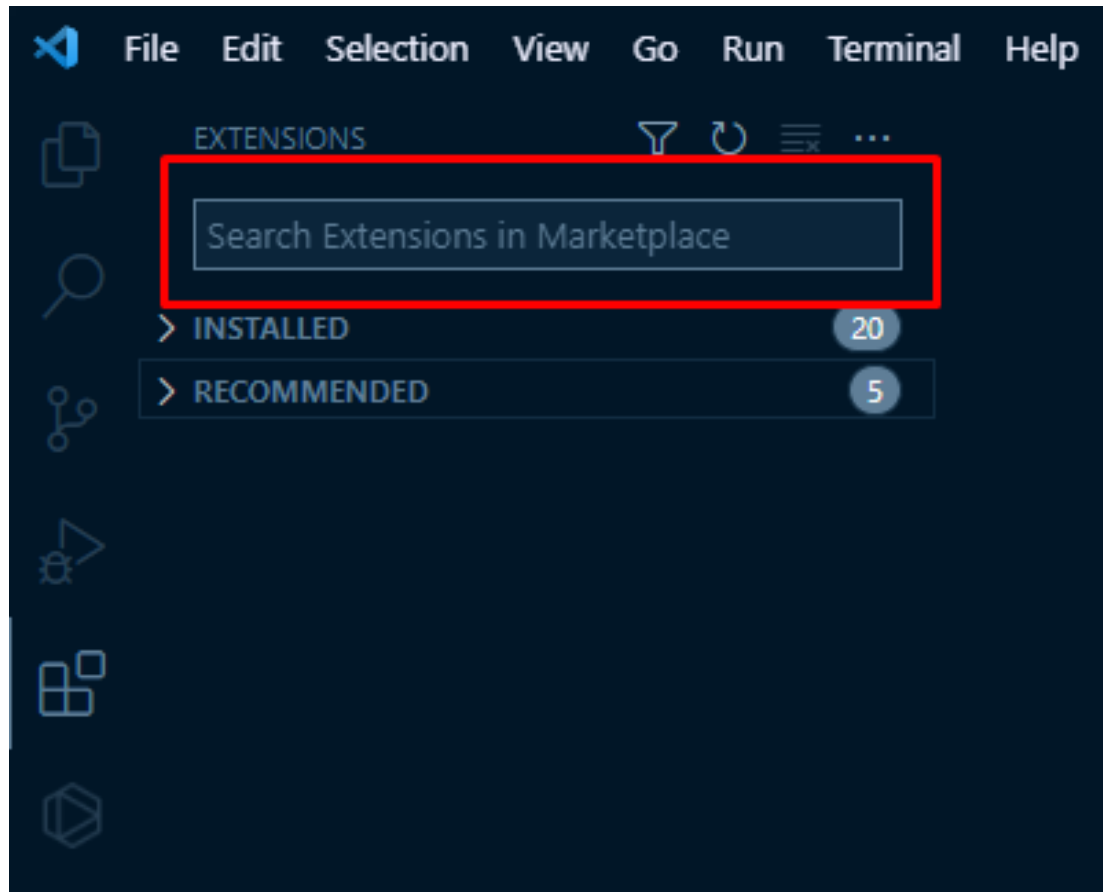
Mientras hago la instalación de Python es muy importante que se tilde la opción de “Add Python 3.X to PATH” como muestra la imagen a continuación. Si no se realiza este paso, es muy probable que tenga problemas de compatibilidad a futuro.



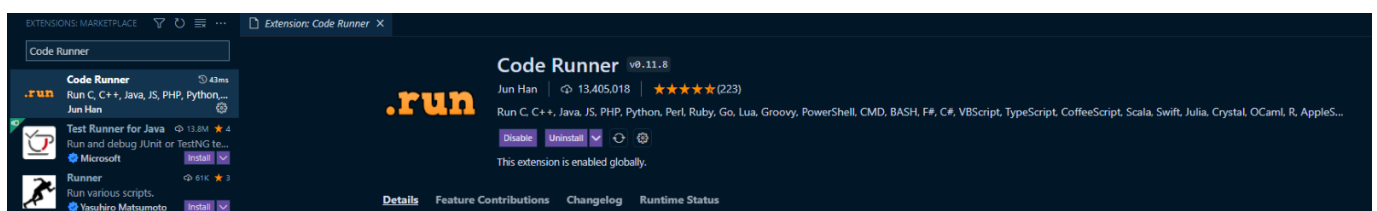
El último paso a la hora de crear el espacio de trabajo es el de poder utilizar Python dentro de Visual Studio Code. Para eso voy a abrir VS Code y me voy a dirigir a la sección de extensiones



Y dentro de esta sección voy a utilizar el buscador indicado a continuación para encontrar e instalar las extensiones que dejo más adelante:

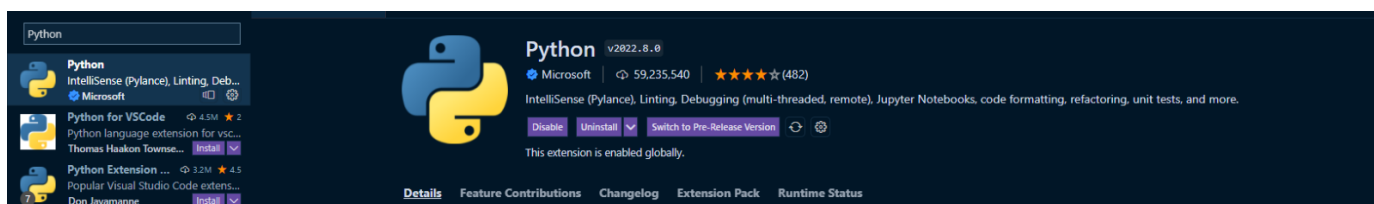


- Code Runner



- Python





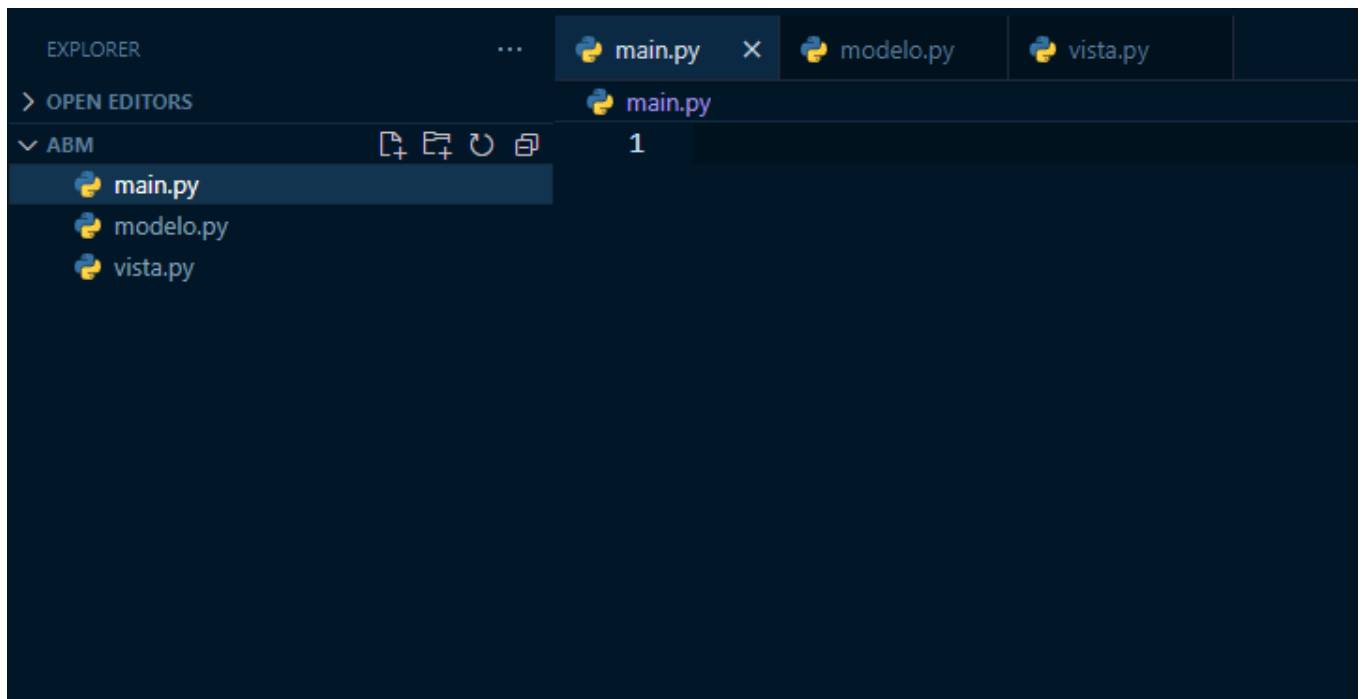
## 2.3 Creación de la estructura MVC

Luego de seguir los anteriores pasos ya debería contar tanto con VS Code como con Python instalados y funcionando en mi computadora así que el siguiente paso es el diseñar la base o arquitectura del programa.

En este caso la arquitectura del sistema va a estar basada en el patrón MVC (Modelo-Vista-Controlador) en el cual cuento con un archivo “Vista” que es por donde el usuario se va a comunicar con el sistema, un archivo “Modelo” que es el que contiene gran parte de la lógica de nuestro sistema y un archivo “Controlador” que se encarga de comunicar a la Vista con el Modelo y que funciona como nuestro módulo principal (El programa se va a ejecutar utilizando este módulo).

Lo primero que hago dentro del Visual Studio Code es crear tres archivos que compondrán al modelo MVC y los titulo de la siguiente manera:

- “main.py” como nuestro Controlador
- “modelo.py” como nuestro Modelo
- “Vista.py” como nuestra Vista



## 2.4 Desarrollo del apartado visual (vista.py)

Voy a separar la parte visual del programa en tres partes para una mejor comprensión del código. Esta división se encuentra graficada a continuación:

Proyecto Analisis de Sistema/POO

## Sistema ABM Libreria

(1)

Titulo

Precio \$

Editorial

Genero

Autor

ID	Titulo	Precio	Editorial	Genero	Autor	Ult. Actualizacion
1	Harry Potter	\$3000	Salamandra	Fantasia	J. K. Rowling	02/07/2022 19:34:28
2	Sobreviviente	\$899	Dunken	Belico	Hernan Arnes	02/07/2022 19:34:31
3	JoJos Bizarre Adventure	\$1200	IVREA	Manga	Hirohiko Araki	02/07/2022 19:34:33

(2)

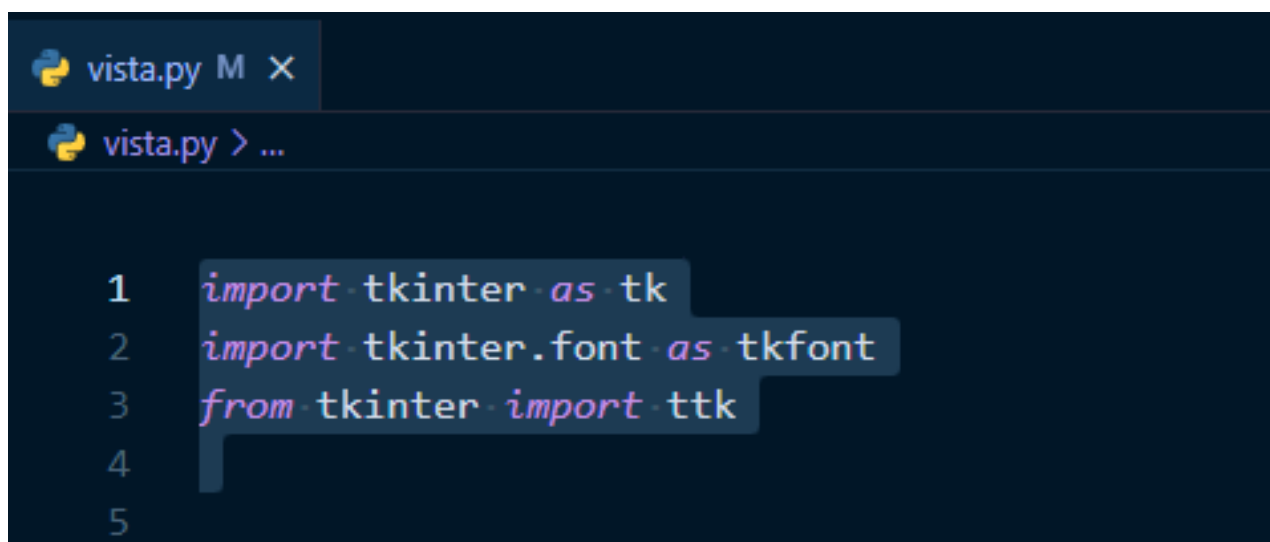
(3) Buscar por titulo

- (1) Sector principal
- (2) Listado de Registros
- (3) Sector de búsqueda/filtro

### 2.4.1 Sector Principal

Abro el archivo “vista.py” en donde voy a importar tres librerías que traen varias funcionalidades para el desarrollo del apartado visual del sistema. Las librerías a importar son las siguientes:

- Tkinter: Es el paquete más utilizado para crear interfaces gráficas en Python, es la herramienta que nos permite crear las etiquetas, botones, campos de texto y hasta la misma ventana donde el usuario interactúa con el sistema.
- Tkfont: Utilizada para modificar la fuente del texto dentro de los elementos de Tkinter.
- Tkinter.Ttk: extensión del paquete Tkinter que me permite crear el listado registros



```
1 import tkinter as tk
2 import tkinter.font as tkfont
3 from tkinter import ttk
4
5
```

Luego voy a crear una clase con nombre “Panel” que tendrá un método de instancia (método que se llama al momento de instanciar a la clase) que recibe los siguientes parámetros:

- self: Hace referencia a la instancia del objeto
- ventana: Hace referencia a un objeto “tk” que va a servir como la ventana donde se posicionan los elementos visuales

- objeto\_modelo: Hace referencia a un objeto de la clase “Abmc” que voy a crear más adelante. Me permite interactuar con la parte lógica del programa

```
#Clase que se encarga del apartado visual del programa  
class Panel():  
  
    # Metodo de instacia de la clase Panel  
    def __init__(self, ventana, objeto_modelo):
```

Y ahora voy a realizar las configuraciones básicas de la ventana y las declaraciones de las variables que se van a utilizar dentro de la misma:

```
#Clase que se encarga del apartado visual del programa  
class Panel():  
  
    # Metodo de instacia de la clase Panel  
    def __init__(self, ventana, objeto_modelo):  
  
        # Creo una variable de instancia root y la cargo con nuestro objeto tk  
        "ventana" que determina la ventana sobre la que estamos trabajando  
        self.root = ventana  
  
        # Configuro el titulo de la ventana sobre la que trabajo  
        self.root.title("Proyecto Analisis de Sistema/P00")  
  
        # Este metodo hace que el tamaño de la ventana no se pueda modificar en  
        ninguno de los dos ejes  
        self.root.resizable(False, False)  
  
        # Declaro y cargo una variable para almacenar el color de fondo del  
        sistema
```

```

self.color_fondo = tk.StringVar()
self.color_fondo.set("#b3cde0")

# Configuro el color de fondo de la ventana utilizando la variable
self.root.configure(bg=self.color_fondo.get())

# Declaracion de variables a utilizar dentro del sistema
# Cada una de estas variables va a ser referenciada dentro de un campo de
texto/Entry
self.var_titulo = tk.StringVar()
self.var_precio = tk.StringVar()
self.var_editorial = tk.StringVar()
self.var_genero = tk.StringVar()
self.var_autor = tk.StringVar()
self.var_id = tk.StringVar()
self.var_busqueda = tk.StringVar()

```

Luego declaro los “tk.Frame”. Este objeto representa un contenedor con un área rectangular en donde se pueden agrupar y organizar los elementos dentro de la ventana del sistema

```

# Declaro los frames donde van a ir ubicado los widgets del sistema
self.frame_central = tk.Frame(
    self.root, bg=self.color_fondo.get())

self.frame_entry = tk.Frame(
    self.frame_central, bg=self.color_fondo.get())

self.frame_entryprecio = tk.Frame(
    self.frame_entry, bg=self.color_fondo.get())

self.frame_botones = tk.Frame(
    self.frame_central, bg=self.color_fondo.get())

self.frame_tree = tk.Frame(

```

```
self.frame_central, bg=self.color_fondo.get())

self.frame_buscar = tk.Frame(
    self.frame_central, bg=self.color_fondo.get())
```

Ahora paso a declarar las Label o Etiquetas principales que sirven como guía para el usuario a la hora de poder ingresar información.

```
# Declaración de las etiquetas que componen al sector inicial de nuestro programa
# El primer parametro es la ubicación en donde se va a posicionar el elemento
# El parametro "bg" determina el color de fondo del elemento
# El parametro "text" determina el texto que va a tener nuestro elemento
# El parametro "font" indica la fuente de este elemento
self.sistema_label = tk.Label(
    self.root,
    bg=self.color_fondo.get(),
    text="Sistema ABM Libreria",
    font=tkfont.Font(family="Times", size=23),
)
self.titulo_label = tk.Label(
    self.frame_entry,
    bg=self.color_fondo.get(),
    text="Titulo",
    font=tkfont.Font(family="Times", size=13),
)
self.editorial_label = tk.Label(
    self.frame_entry,
    bg=self.color_fondo.get(),
    text="Editorial",
    font=tkfont.Font(family="Times", size=13),
)
self.simbolopeso_label = tk.Label(
    self.frame_entryprecio,
    bg=self.color_fondo.get(),
```

```

        text="$",
        font=tkfont.Font(family="Times", size=13),
    )
    self.precio_label = tk.Label(
        self.frame_entryprecio,
        bg=self.color_fondo.get(),
        text="Precio",
        font=tkfont.Font(family="Times", size=13),
    )
    self.genero_label = tk.Label(
        self.frame_entry,
        bg=self.color_fondo.get(),
        text="Genero",
        font=tkfont.Font(family="Times", size=13),
    )
    self.autor_label = tk.Label(
        self.frame_entry,
        bg=self.color_fondo.get(),
        text="Autor",
        font=tkfont.Font(family="Times", size=13),
    )

```

Creo los campos de entrada que le permiten ingresar datos en el sistema al usuario y van a estar ubicados al lado de sus etiquetas/labels correspondientes

```

# Declaración de las campos de texto que componen al sector inicial de
nuestro programa
# El primer parametro es la ubicacion en donde se va a pocisionar el
elemento
# El parametro "textvariable" determina la variable en donde se va a
almacenar la informacion de nuestro Entry
# El parametro "font" indica la fuente de este elemento
    self.titulo_entry = tk.Entry(
        self.frame_entry,

```



```

        textvariable=self.var_titulo,
        font=tkfont.Font(family="Times", size=13),
    )

    self.precio_entry = tk.Entry(
        self.frame_entry,
        textvariable=self.var_precio,
        font=tkfont.Font(family="Times", size=13),
    )

    self.editorial_entry = tk.Entry(
        self.frame_entry,
        textvariable=self.var_editorial,
        font=tkfont.Font(family="Times", size=13),
    )

    self.gen_entry = tk.Entry(
        self.frame_entry,
        textvariable=self.var_genero,
        font=tkfont.Font(family="Times", size=13),
    )

    self.autor_entry = tk.Entry(
        self.frame_entry,
        textvariable=self.var_autor,
        font=tkfont.Font(family="Times", size=13),
    )

```

Creo los botones principales del programa que son “Guardar”, “Actualizar” y “Eliminar”. Cada uno de estos botones va a llamar a una función que voy a declarar más adelante en el módulo “Modelo” donde se encuentra la lógica del programa

```

# Botones del sector principal
#El primer parametro de cada boton determina su ubicacion

```

```
#El parametro "state" determina si el boton es clickeable o no
```

```
#Boton Eliminar
```

```
#Llama a la funcion "funcion_baja" que crearemos mas adelante en el modulo "modelo"
```

```
self.borrar_boton = tk.Button(  
    self.frame_botones,  
    command=lambda: objeto_modelo.funcion_baja(  
        self.var_id.get(),  
        self.upd_boton,  
        self.borrar_boton,  
        self.tree,  
        self.var_titulo,  
        self.var_precio,  
        self.var_editorial,  
        self.var_genero,  
        self.var_autor,  
        self.var_busqueda,  
    ),  
    font=tkfont.Font(family="Times", size=13),  
    text="Eliminar",  
    state="disabled",  
)
```

```
# Boton guardar
```

```
# Llama a la funcion "funcion_alta" que crearemos mas adelante en el modulo "modelo"
```

```
self.g_boton = tk.Button(  
    self.frame_botones,  
    command=lambda: objeto_modelo.funcion_alta(  
        self.var_titulo,  
        self.var_precio,  
        self.var_editorial,  
        self.var_genero,  
        self.var_autor,
```

```
        self.upd_boton,  
        self.borrar_boton,  
        self.var_busqueda,  
        self.tree,  
    ),  
    font=tkfont.Font(family="Times", size=13),  
    text="Guardar",  
)
```

```
# Boton actualizar
```

```
#Llama a la funcion "funcion_modificar" del modulo "modelo"
```

```
#permite guardar la actualizacion de un registro en la base de datos
```

```
self.upd_boton = tk.Button(  
    self.frame_botones,  
    command=lambda: objeto_modelo.funcion_modificar(  
        self.var_id,  
        self.var_titulo,  
        self.var_precio,  
        self.var_editorial,  
        self.var_genero,  
        self.var_autor,  
        self.upd_boton,  
        self.borrar_boton,  
        self.var_busqueda,  
        self.tree,  
    ),  
    font=tkfont.Font(family="Times", size=13),  
    text="Actualizar",  
    state="disabled",  
)
```

Y el último paso para dentro del sector en donde se hace el ingreso de datos es el de ubicar los elementos que cree previamente. Para eso voy a utilizar el método “grid” que me permite ubicar cada elemento dentro de una grilla imaginaria dentro de la ventana.

Los parámetros “row” y “column” determinan en qué fila y columna se va a posicionar cada elemento de la grilla, mientras que los parámetros “padx” y “pady” permiten mover al elemento dentro del eje X e Y para un posicionamiento más preciso.

```
self.frame_central.grid(row=1, column=0)
self.frame_entry.grid(row=0, column=0)
self.frame_botones.grid(row=1, column=0)
self.frame_tree.grid(row=2, column=0)
self.frame_buscar.grid(row=3, column=0)

self.sistema_label.grid(row=0, column=0)

self.titulo_label.grid(row=1, column=0, padx=15, pady=(20, 5))
self.titulo_entry.grid(row=1, column=1, padx=15, pady=(20, 5))

self.frame_entryprecio.grid(row=2, column=0, padx=(30, 0))
self.precio_label.grid(row=0, column=0, pady=5, padx=(0, 30))
self.simbolopeso_label.grid(row=0, column=1)

self.precio_entry.grid(row=2, column=1)

self.editorial_label.grid(row=3, column=0, pady=5)
self.editorial_entry.grid(row=3, column=1)

self.genero_label.grid(row=4, column=0, pady=5)
self.gen_entry.grid(row=4, column=1)

self.autor_label.grid(row=5, column=0, pady=5)
self.autor_entry.grid(row=5, column=1)
```

```
self.g_boton.grid(row=0, column=0, padx=30, pady=10)
self.upd_boton.grid(row=0, column=1, padx=10, pady=10)
self.borrar_boton.grid(row=0, column=2, padx=30, pady=10)
```

## 2.4.2 Listado de registros

Para la creación del listado voy a usar el objeto “Treeview” perteneciente a la librería “tk”.

Un dato a aclarar es que los treeview vienen con una columna por default (normalmente llamada “ghost column” o “columna fantasma”) y voy a utilizar esa columna para almacenar el “ID” de cada elemento, por eso no se declara la columna “ID” dentro del array de columnas “self.columns”.

```
#Declaro un array con nombres para identificar a cada columna
self.columns = ("titulo", "precio", "editorial",
               "genero", "autor", "fecha_upd")

#Creo el listado treeview y le paso como parametros la ubicacion
en donde va a estar posicionado
#y el listado de las columnas
self.tree = ttk.Treeview(self.frame_tree, columns=self.columns)

#Declaro los heading o encabezados de la lista y le asigno un
nombre
self.tree.heading("#0", text="ID")

#Modifico la configuracion de la columna correspondiente a cada
encabezado

#-minwidth determina el ancho minimo que puede tener la columna
#-width determina el ancho predeterminado de la columna
#-anchor determina como ubico el texto dentro de la columna, en
este caso el texto estaria centrado
```

```

self.tree.column("#0", minwidth=0, width=40, anchor="center")

self.tree.heading("titulo", text="Titulo")
self.tree.column("titulo", minwidth=0, width=150, anchor="center")

self.tree.heading("precio", text="Precio")
self.tree.column("precio", minwidth=0, width=50, anchor="center")

self.tree.heading("editorial", text="Editorial")
self.tree.column("editorial", minwidth=0, width=100,
anchor="center")

self.tree.heading("genero", text="Genero")
self.tree.column("genero", minwidth=0, width=100, anchor="center")

self.tree.heading("autor", text="Autor")
self.tree.column("autor", minwidth=0, width=100, anchor="center")

self.tree.heading("fecha_upd", text="Ult. Actualizacion")
self.tree.column("fecha_upd", minwidth=0, width=120,
anchor="center")

#Pocisiono el listado tree dentro de la grilla
self.tree.grid(row=0, column=0, padx=30)

```

Luego creo un evento utilizando el método “bind” que me permite ejecutar una función cada vez que el usuario hace click en el listado

```

# Evento que se acciona cuando hago click en un item del treeview
self.tree.bind(
    "<<TreeviewSelect>>",
    lambda event: objeto_modelo.select_item(
        self.tree.item(self.tree.focus()),
        self.var_id,

```

```
        self.var_titulo,  
        self.var_precio,  
        self.var_editorial,  
        self.var_genero,  
        self.var_autor,  
        self.borrar_boton,  
        self.upd_boton,  
    ),  
)
```

Y por último llamo a un método que voy a crear más adelante y que me permite mostrar los registros ingresados dentro del listado Treeview.

```
# Llamo a funcion para popular la lista  
# Esta funcion solo se llama una vez  
objeto_modelo.cargar_listado(self.tree, self.var_busqueda)
```

## 2.4.3 Sector de búsqueda/filtro

El último sector a desarrollar en el módulo de vista es el de búsqueda/filtro. Este sector está compuesto por:

- Etiqueta/Label: identificador del campo de entrada .
- Campo de texto: espacio en donde el usuario puede ingresar el título del producto a buscar.
- Botón “Buscar”: ejecuta un función “cargar\_listado” del módulo “modelo” que modifica la información del listado treeview dependiendo de los datos ingresados en el campo de texto.
- Botón “Borrar Filtro”: ejecuta la misma función que el botón “Buscar” pero aplicando una lógica distinta que permite vaciar

los campos de entrada y mostrar el listado de registros completos sin aplicar filtros.

Así debería verse el código que compone al “sector búsqueda” del sistema:

```
#Label "Buscar por titulo"
self.buscar_label = tk.Label(
    self.frame_buscar,
    bg=self.color_fondo.get(),
    text="Buscar por titulo",
    font=tkfont.Font(family="Times", size=13),
)

#Campo de texto para ingresar titulo utilizado en el filtro
self.buscar_entry = tk.Entry(
    self.frame_buscar,
    textvariable=self.var_busqueda,
    font=tkfont.Font(family="Times", size=13),
)

#Boton de busqueda: muestra solo los registros que contengan el
texto del entry "buscar_entry" dentro del listado de registros
self.busq_boton = tk.Button(
    self.frame_buscar,
    command=lambda: objeto_modelo.cargar_listado(
        self.tree,
        self.var_busqueda,
    ),
    font=tkfont.Font(family="Times", size=13),
    text="Buscar",
)

#Boton borrar filtro: vacia campos de texto y reinicia el listado
de registros
```



```

self.b_filtro_boton = tk.Button(
    self.frame_buscar,
    command=lambda: objeto_modelo.cargar_listado(
        self.tree,
        self.var_busqueda,
        True,
    ),
    font=tkfont.Font(family="Times", size=13),
    text="Borrar filtro",
)

#Pocisionamiento de la etiquetas, campos de entrada y los botones
del sector busqueda
self.buscar_label.grid(row=8, column=0, padx=10, pady=10)
self.buscar_entry.grid(row=8, column=1, pady=10)

self.busq_boton.grid(row=8, column=2, padx=20, pady=10)
self.b_filtro_boton.grid(row=8, column=3, padx=10, pady=10)

```

Con esto ya doy por finalizado el desarrollo del apartado visual del programa. Ahora paso a crear la lógica del sistema que me va a permitir procesar la información que ingrese el usuario desde el apartado visual.

## 2.5 Lógica del sistema (modelo.py)

El primer paso a la hora de desarrollar la lógica del programa es el de abrir el archivo “modelo.py” previamente creado en el inicio de

esta guía. Dentro de este módulo voy a estar declarando las siguientes librerías:

```
#Base de datos que utilizo para almacenar registros
import sqlite3
#Libreria que permite utilizar expresiones regulares para validar
informacion
import re
#Libreria que utilizo para conseguir la fecha y hora exacta al momento de
registrar cambios en la base de datos
from datetime import datetime
#Libreria utilizada para mostrar popups o mensajes de alerta a la hora de
realizar algun cambio en el registro de datos
from tkinter import messagebox
```

Luego creo la clase “Abmc” y dentro de su método de instancia (“def \_\_init\_\_(self,)”) voy a crear tres variables “regex” que me van a servir a la hora de realizar validaciones. Para tener una idea del funcionamiento de las “regex” o “expresiones regulares” dejo este link a continuacion:

<https://www.ionos.es/digitalguide/paginas-web/creacion-de-paginas-web/regex/>

```
#Clase ABMC: maneja la logica relacionada a las funciones de Alta, Baja,
Modificacion y Consulta del sistema
class Abmc():
    #Metodo de instancia
    def __init__(self,):
        # Matchea con lineas de texto que solo tengan numeros entre 0 y 9
        self.regex_numeros = "^[0-9]+$"

        # Matchea con lineas de texto que solo tengan caracteres y con una
        longitud entre 1 y 60 caracteres
```

```

self.regex_palabra = r"^[.,a-zA-Z\s]{1,60}$"

# Matchea con lineas de texto que tengan caracteres o numeros y
con una longitud entre 1 y 60 caracteres
self.regex_vacio = r"^[.,a-zA-Z0-9_\s]{1,60}$"

```

## Todas las funciones que se creen a continuación van a estar dentro de la clase “Abmc”

“crear\_db”: permite conectarse/crear a la base de datos SQLite3 y crear la tabla “libros” dentro de esta base:

```

#Funcion que devuelve una conexion a La base de datos
def crear_db(self,):
    #El metodo "connect" de sqlite3 me permite conectarme a una base
de datos con el nombre que es enviado como parametroo
    #En el caso de que haya una base de datos con ese nombre en el
mismo directorio que el archivo, este metodo nos conecta a esa base de
datos
    #En el caso de que NO haya una base de datos con ese nombre, este
metodo crea la base de datos y luego nos conecta
    basedatos = sqlite3.connect("libreria.db")
    #Devuelvo la referencia a la base de datos
    return basedatos

#Funcion para crear la tabla de registros de libros en nuestra base de
datos
#Toma como parametro a la base de datos que estamos utilizando
def crear_tabla_libro(self, basedatos):

    #Consigo la referencia a la base de datos utilizando la funcion de
conexion
    basedatos = self.crear_db()

```

```

        #Creo un objeto cursor que me permite ejecutar sentencias SQL en
la base de datos
        cursorbdd = basedatos.cursor()

        #Defino una sentencia SQL que me permite crear la tabla "Libros"
si es que no fue creada con anterioridad
        sql = (
            "CREATE TABLE IF NOT EXISTS libros(id INTEGER PRIMARY KEY
AUTOINCREMENT,"
            " titulo TEXT,"
            " precio INTEGER,"
            " editorial TEXT,"
            " genero TEXT,"
            " autor TEXT,"
            " fecha_mod DATETIME)"
        )

        #Ejecuta la sentencia utilizando el cursor
        cursorbdd.execute(sql)

        #Utilizo el metodo commit para guardar los cambios realizados en
la base de datos
        basedatos.commit()

```

“toggle\_botones”: me permite habilitar o deshabilitar los botones de actualizar y eliminar según sea necesario. La idea de esta función es hacer que los botones previamente mencionados sean clickeables solo si el usuario hizo click en un elemento de la lista, en caso contrario, solo el botón de “Agregar” debería poder ser seleccionado.

```

        # Funcion para determinar si los botones de actualizacion y borrado se
encuentran activos

```

```

# Esta funcion toma como parametros a Los objetos boton de
actualizacion y boton de borrado; tambien toma un parametro booleano
deshabilitar

def toggle_botones(self, upd_boton, borrar_boton, deshabilitar):
    #Si dehabilitar es "True", deshabilitamos Los botones de
actualizacion y borrado
    if deshabilitar:
        # "state" es un atributo de Los botones de Tkinter que
determina si el boton es clickeable o no
        upd_boton["state"] = "disabled"
        borrar_boton["state"] = "disabled"
    #En el caso de que deshabilitar sea "False", hago que Los botones
sean clickeables
    else:
        upd_boton["state"] = "normal"
        borrar_boton["state"] = "normal"

```

“vaciar\_entradas”: resetea los campos de entrada de texto

```

# Funcion para vaciar todos Los campos de entrada luego de realizar
una alta, baja o modificacion
# Toma como parametro a Los entry del sector principal del programa
def vaciar_entradas(self, titulo, precio, editorial, genero, autor):
    #El metodo "set("")" nos permite vaciar cada uno de Los campos de
entrada
    titulo.set("")
    precio.set("")
    editorial.set("")
    genero.set("")
    autor.set("")

```

“select\_item”: permite recibir los datos de los registros seleccionados en el listado y a su vez llena los campos de entrada con la información recibida

```

        # Funcion para rellena los entry con datos del item seleccionado en
treeview
        # Toma como parametro al objeto listado, los campos de entrada y los
botones de borrar y actualizar
        def select_item(
            self, lista, id, titulo, precio, editorial, genero, autor,
borrar_boton, upd_boton
        ):

            try:
                # Rellenamos los campos de entrada con los valores del item
seleccionado dentro de la lista
                id.set(lista["text"])
                titulo.set(lista["values"][0])
                precio.set(lista["values"][1].lstrip("$"))
                editorial.set(lista["values"][2])
                genero.set(lista["values"][3])
                autor.set(lista["values"][4])
            except IndexError:
                #En el caso de que haya un error a la hora de conseguir los
valores del item, se sale de la funcion
                return

            # Habilito los botones de actualizar y eliminar ya que estoy
seleccionando un registro
            self.toggle_botones(upd_boton, borrar_boton, False)

```

“validar\_entrada”: valida los datos ingresados en los campos de entrada a la hora de añadir, eliminar o actualizar un registro de la base de datos. La validación se hace utilizando las expresiones regulares declaradas al principio de nuestra clase “Abmc”

```

# Funcion para validar los campos de entrada de dato teniendo en
cuenta diferentes criterios

# Toma como parametro a los campos de entrada del sector principal
def validar_entrada(self, titulo, precio, editorial, genero, autor):

    # Cargo los patrones de regex que voy a usar para validar cada
campo

    patron_num = re.compile(self.regex_numeros)
    patron_text = re.compile(self.regex_palabra)
    patron_vacio = re.compile(self.regex_vacio)

    # Variable en la que concateno cada campo que tenga un error
errores = ""

    #el metodo fullmatch de la libreria regex permite comparar un
string con una expresion regular

    #En este caso estoy verificando si mi string no cumple con las
condiciones de la expresion regular
    if not re.fullmatch(patron_vacio, titulo.get()):
        #En el caso de que no cumpla con esa condicion, agregamos el
nombre del campo de entrada a nuestro
        #string de errores
        errores += "\n-Titulo"
    if not re.fullmatch(patron_num, precio.get()):
        errores += "\n-Precio"
    if not re.fullmatch(patron_text, editorial.get()):
        errores += "\n-Editorial"
    if not re.fullmatch(patron_text, genero.get()):
        errores += "\n-Genero"
    if not re.fullmatch(patron_text, autor.get()):
        errores += "\n-Autor"

    # Si la variable error no esta vacia, muestra un mensaje indicando
los campos que no estan ingresados correctamente
    if errores != "":

```

```

        #El metodo showerror de la libreria messagebox permite mostrar
una ventana emergente con los parametros ingresados
        messagebox.showerror(
            title="Error",
            message="Los siguientes campos no se ingresaron
correctamente: " + errores,
        )
        return True

```

“cargar\_listado”: carga el listado treeview con los elementos presentes en la base de datos. El comportamiento de la función cambia si el campo de filtro fue cargado con datos previamente

```

# Funcion para poblar el listado treeview
# Toma como parametros al listado, el campo de entrada de filtro y una
variable "borrar_filtro" que se utiliza
# Para determinar si se hizo click en el boton de borrar filtro
def cargar_listado(
    self, tree, filtro, borrar_filtro=""
):

    # Me conecto a la base de datos
    basedatos = self.crear_db()
    # Creo un cursor apuntando a la base de datos
    cursorbdd = basedatos.cursor()

    # Cargo una variable con el contenido del campo de entrada de
filtro
    busqueda = filtro.get()

    # El atributo borrar_filtro es una bool
    # Lo utilizo para determinar si se toco el boton de borrar filtros

```



```

        # En el caso de que el campo busqueda este vacio o si se hizo
click en el boton de borrar filtros
    if busqueda == "" or borrar_filtro:
        # Vacía el campo de entrada de filtro
        filtro.set("")
        # Trae el listado completo de registros sin filtrar
        cursorbdd.execute("SELECT * FROM libros")
        # Creo un listado con los registros traídos de la base de
datos
        lista_datos = cursorbdd.fetchall()
    else:
        # En el caso de que se haya ingresado información en el campo
de entrada de filtro

        # Cargo una variable con la información agregada en el filtro
        datos = (busqueda,)
        # Con esta declaración puedo traer todos los registros de la
base de datos que contengan el título que busco
        # Utilizamos los caracteres "?" para hacer referencia a un
parametro que vamos a indicar más adelante
        sql_select = "SELECT * FROM libros WHERE titulo LIKE '%' || ?
|| '%' "
        # Ejecuto la secuencia SQL
        # El array "datos" contiene el parametro que va a reemplazar el
símbolo "?" de la secuencia "sql_select"
        cursorbdd.execute(sql_select, datos)
        # Creo un listado con los registros traídos de la base de
datos
        lista_datos = cursorbdd.fetchall()

        # Borra todos los registros del tree view
        for i in tree.get_children():
            tree.delete(i)

        # Recorre el listado de registros y los introduce en el treeview
        for item in lista_datos:

```

```

        # Formateo la fecha para mostrarla en el formato de dd/mm/yyyy
h:m:s

        fecha = datetime.strptime(str(item[6]), "%Y/%m/%d %H:%M:%S")
        fecha_format = datetime.strftime(fecha, "%d/%m/%Y %H:%M:%S")

        # Agrego un registro en el listado por cada item traído de la
base de datos

        tree.insert(
            "",
            "end",
            text=str(item[0]),
            values=(
                item[1],
                "$" + str(item[2]),
                item[3],
                item[4],
                item[5],
                fecha_format,
            ),
        )

```

“funcion\_baja”: permite eliminar un registro de la base de datos luego de haber sido seleccionado en la lista

```

# Funcion para dar de baja registros de la base de datos
# Toma como parametros al id del registro a eliminar, al objeto de
base de datos
# y a los campos de entrada del sector principal y de busqueda
def funcion_baja(
    self,
    id,
    upd_boton,

```

```

        borrar_boton,
        tree,
        titulo,
        precio,
        editorial,
        genero,
        autor,
        busqueda,
    ):

        # Me conecto a La base de datos
        basedatos = self.crear_db()
        # Creo un cursor apuntando a la base de datos
        cursorbdd = basedatos.cursor()

        # En el caso de que el campo Id este vacio
        if id == "":
            # Se muestra un mensaje emergente indicando que no se
            selecciono ningun registro del listado
            messagebox.showinfo(
                title="Info", message="No se ha seleccionado ningun
registro en el listado"
            )
            # Salgo de La funcion
            return

        # Muestro un mensaje de consulta al usuario
        if messagebox.askyesno(
            title="Confirmar eleccion", message="¿Desea eliminar el
registro de la lista?"
        ):
            # En el caso de que el usuario haga click en el boton "Si"
            dentro del mensaje emergente ->

            # Secuencia SQL para eliminar el registro con el mismo id al
            ingresado

```

```

        sql_delete = "DELETE FROM libros WHERE id = ?"
        datos = (id,)
        cursorbdd.execute(sql_delete, datos)
        # Guardo los cambios en la base de datos
        basedatos.commit()
        # Muestro un mensaje indicando que el proceso ha finalizado
con exito
        messagebox.showinfo(
            title="Info", message="Registro eliminado con exito")

        # Refresco el treeview luego de hacer un cambio
        self.cargar_listado(tree, busqueda)
        # Deshabilito el uso de los botones
        self.toggle_botones(upd_boton, borrar_boton, True)
        # Vacio los campos de entrada
        self.vaciar_entradas(titulo, precio, editorial, genero, autor)

```

“funcion\_alta”: crea un registro en la base de datos utilizando los datos ingresados en el sector principal del programa

```

def funcion_alta(
    self, titulo, precio, editorial, genero, autor, upd_boton,
borrar_boton, busqueda, tree
):
    # Me conecto a la base de datos
    basedatos = self.crear_db()
    # Creo un cursor apuntando a la base de datos
    cursorbdd = basedatos.cursor()

    # Consigo la fecha y hora actual
    horario_actual = datetime.now().strftime("%Y/%m/%d %H:%M:%S")

    # Sale de la funcion si algun dato ingresado es incorrecto
    if self.validar_entrada(titulo, precio, editorial, genero, autor):
        return

```

```

# Creo una tupla con los datos ingresados en los campos de entrada
datos = (
    titulo.get(),
    precio.get(),
    editorial.get(),
    genero.get(),
    autor.get(),
    horario_actual,
)

# creo una secuencia SQL que recibe como parametro al listado de
datos ingresados
sql_insert = "INSERT INTO libros (titulo, precio, editorial,
genero, autor, fecha_mod) VALUES (?, ?, ?, ?, ?, ?)"
# Ejecuto la secuencia y guardo el registro
cursorbdd.execute(sql_insert, datos)
basedatos.commit()
# Muestro un mensaje indicando que el proceso ha finalizado con
exito
messagebox.showinfo(
    title="Info", message="Registro ingresado con exito!")

# Refresco el treeview luego de hacer un cambio
self.cargar_listado(tree, busqueda)
# Vacio los campos de entrada
self.vaciar_entradas(titulo, precio, editorial, genero, autor)
# Deshabilito el uso de los botones
self.toggle_botones(upd_boton, borrar_boton, True)

```

“Funcion\_modificar”: modifica un registro ya presente en la base de datos utilizando los datos ingresados en el sector principal del programa

```

# Funcion para modificar registros
# Toma como parametros a los campos de entrada, el listado de
registros y los botones de actualizar y eliminar
def funcion_modificar(
    self,
    id_registro,
    titulo,
    precio,
    editorial,
    genero,
    autor,
    borrar_boton,
    upd_boton,
    busqueda,
    tree,
):

    # Me conecto a La base de datos
    basedatos = self.crear_db()
    # Creo un cursor apuntando a La base de datos
    cursorbdd = basedatos.cursor()

    # Verifico si se selecciono algun registro del listado
    if id_registro.get() == "":
        messagebox.showinfo(
            title="Info", message="No se ha seleccionado ningun
registro en el listado"
        )
        return

    # Valido los datos ingresados
    elif self.validar_entrada(titulo, precio, editorial, genero,
autor):
        return

    # Consigo la fecha y hora actual
    horario_actual = datetime.now().strftime("%Y/%m/%d %H:%M:%S")

```

```

# Cargo una tupla con los datos a ingresar
datos = (
    titulo.get(),
    precio.get(),
    editorial.get(),
    genero.get(),
    autor.get(),
    horario_actual,
    id_registro.get(),
)

# Muestro un mensaje de consulta al usuario
if messagebox.askyesno(
    title="Confirmar eleccion", message="¿Desea modificar el
registro de la lista?"
):
    # Modifica la informacion del registro que tenga el mismo id
del item seleccionado en el listado
    sql_update = (
        "UPDATE libros SET titulo = ?, precio =?, editorial =?,
genero = ?, autor = ?, fecha_mod = ? "
        "WHERE id = ?"
    )
    # Ejecuto la secuencia SQL y guardo los cambios en la base de
datos
    cursorbdd.execute(sql_update, datos)
    basedatos.commit()
    # Muestro un mensaje indicando que el proceso ha finalizado
con exito
    messagebox.showinfo(
        title="Info", message="Registro modificado con exito")

# Refresco el treeview luego de hacer un cambio
self.cargar_listado(tree, busqueda)
# Vacio los campos de entrada

```

```
self.vaciar_entradas(titulo, precio, editorial, genero, autor)
# Deshabilito el uso de los botones
self.toggle_botones(upd_boton, borrar_boton, True)
```

## 2.6 Uniendo el apartado visual y la logica (main.py)

Este es el ultimo paso en el desarrollo del programa en el cual voy a utilizar a mi modulo “main.py” como controlador de mi sistema MVC permitiendo comunicar a la vista (parte visual) y al modelo (parte lógica).

Lo primero que hago es abrir el archivo “main.py” y llamo a las siguientes librerias:

```
#Importo la Librería TK para poder crear una ventana que luego voy a
utilizar a la hora de llamar a mi Vista
from tkinter import Tk
#Importo el modulo "modelo" para poder utilizar sus clases y funciones
import modelo
#Importo el modulo "vista" para poder utilizar sus clases y funciones
import vista
```

Y creo la clase “Controlador” que interactúa con el modulo de “modelo” y “vista”:

```
#Creo la clase controlador
class Controller:
    # funcion que se ejecuta al instanciar la clase
    # Toma como parametro a un objeto Tk (ventana del programa)
    def __init__(self, ventana, ):
```



```

        #Creo un objeto utilizando la clase "Abmc" del modulo "modelo"
        self.objeto_modelo = modelo.Abmc()
        #Creo/Conecto la base de datos
        self.objeto_modelo.crear_db()
        #Creo la tabla "Libros" en caso de que no haya sido creada
        previamente
        self.objeto_modelo.crear_tabla_libro()
        # guardo la ventana en un atributo de instancia
        self.ventana = ventana
        # Creo un objeto "Panel" de modulo vista y le envio la ventana
        como parametro
        self.objeto_vista = vista.Panel(self.ventana, self.objeto_modelo)

```

Por último creó una condición que verifique si el módulo en donde estoy parado está siendo ejecutado como mi programa principal. Dentro de esta condición se crean las instancias de la ventana y del controlador de la aplicación.

```

# verifica que el modulo se haya ejecutado como programa principal
if __name__ == "__main__":

    #Creo la ventana donde voy a pocisionar mis elementos visuales
    haciendo una instacia del objeto "TK"
    root = Tk()

    # Instancio un objeto "Controller" y le envio la ventana como
    parametro
    mi_app = Controller(root)

    # Crea un loop infinito que permite visualizar la ventana
    # Sin esta funcion, la ventana se cerraria al momento de ejecutar el
    programa
    root.mainloop()

```

Con esto hecho el programa ya debería estar funcionando sin problemas. Cabe aclarar que para que el programa funcione hay que ejecutarlo si o si desde el archivo “main.py”.