

PYTHON

APP ESCRITORIO CON
THINKER Y BBDD

PRÁCTICA 2-M6



Práctica de creación de una app de escritorio con base de datos

En esta práctica se va a crear una app de escritorio con conexión a una base de datos. Esta aplicación tendrá como objetivo ser un gestor de productos, es decir, una aplicación que permitirá al usuario realizar las siguientes acciones:

- Crear un producto (nombre y precio).
- Editar un producto.
- Eliminar un producto.

El stack tecnológico que se usará en este proyecto es el siguiente:

- Python 3. Como lenguaje de programación base.
- JetBrains Pycharm Community. IDE escogido para el desarrollo del proyecto.
- Tkinter. Módulo integrado en Python que proporciona interfaces gráficas.
- SQLite. Base de datos SQL rápida y potente para instalaciones de tamaño moderado.
- Virtualenv. Entorno virtual de Python donde se programará el proyecto.

Pre requisitos del proyecto:

- Python 3 instalado.
- Pycharm instalado.
- SQLite instalado.

El objetivo de la práctica no es crear una aplicación estéticamente bonita, ya que Tkinter prevalece en sencillez frente a la estética. El objetivo es crear una aplicación de gestión, sencilla, útil y funcional.

Es imposible abordar una explicación completa del módulo Tkinter en este proyecto, por lo tanto, en la bibliografía se encuentra el enlace a la documentación oficial (muy técnica) y a una guía (no oficial) que explica la estructura y componentes gráficos de Tkinter en español.

Para terminar con esta introducción, veamos un pantallazo del resultado de este proyecto:



App Gestor de Productos

Registrar un nuevo Producto

Nombre:

Precio:

Guardar Producto

Nombre	Precio
Arduino	20.0
Dron	300.0
Impresora 3D	1000.0
Raspberry Pi 4	60.0

ELIMINAR **EDITAR**

App Gestor de Productos

Edición de Productos

Editar el siguiente Producto

Nombre antiguo: Raspberry Pi 4

Nombre nuevo:

Precio antiguo: 60.0

Precio nuevo:

Actualizar Producto

DB Browser for SQLite - C:\Users\Cristian\PycharmProjects\Ges

Archivo Editar Ver Herramientas Ayuda

Nueva base de datos Abrir base de datos Guardar

Estructura Hoja de datos Editar pragmas Ejecutar SQL

Tabla: producto

	id	nombre	precio
	Filtro	Filtro	Filtro
1	1	Impresora 3D	1000.0
2	2	Arduino	20.0
3	3	Raspberry Pi 4	60.0
4	4	Dron	300.0



Contenido

1.	Creación del proyecto e integración en un entorno virtual	6
2.	Instalación de módulos dentro del entorno virtual	10
3.	Salir y entrar del entorno virtual	11
4.	Creación del fichero Python principal y primera ventana	12
5.	Comenzar la implementación usando Clases y Objetos	15
6.	Configuración base de la ventana principal	16
7.	Estructura de los widgets (componentes gráficos) en ventana.....	18
8.	Widgets utilizados en este proyecto	21
9.	Posicionamiento de los elementos en Tkinter	23
10.	Comencemos con la interfaz gráfica. Añadir Producto	28
11.	Interfaz gráfica. Tabla de Productos	32
12.	Creación de la base de datos.....	34
13.	Inserción de datos de prueba en la base de datos.....	39
14.	Implementar la conexión a la base de datos desde Python	41
15.	Implementar el método de get_productos	43
16.	Implementar el método de add_producto() y sus validaciones	47
17.	Mejorando add_producto()	51
18.	Añadir los dos botones que falta: Eliminar y Editar	55
19.	Implementar la funcionalidad de Eliminar.....	56
20.	Implementar la funcionalidad de Editar.....	63
21.	Mejorando el diseño.....	73
22.	Resultado final.....	75
23.	Mejoras y entrega de la práctica	78
24.	Bibliografía.....	79



1. Creación del proyecto e integración en un entorno virtual

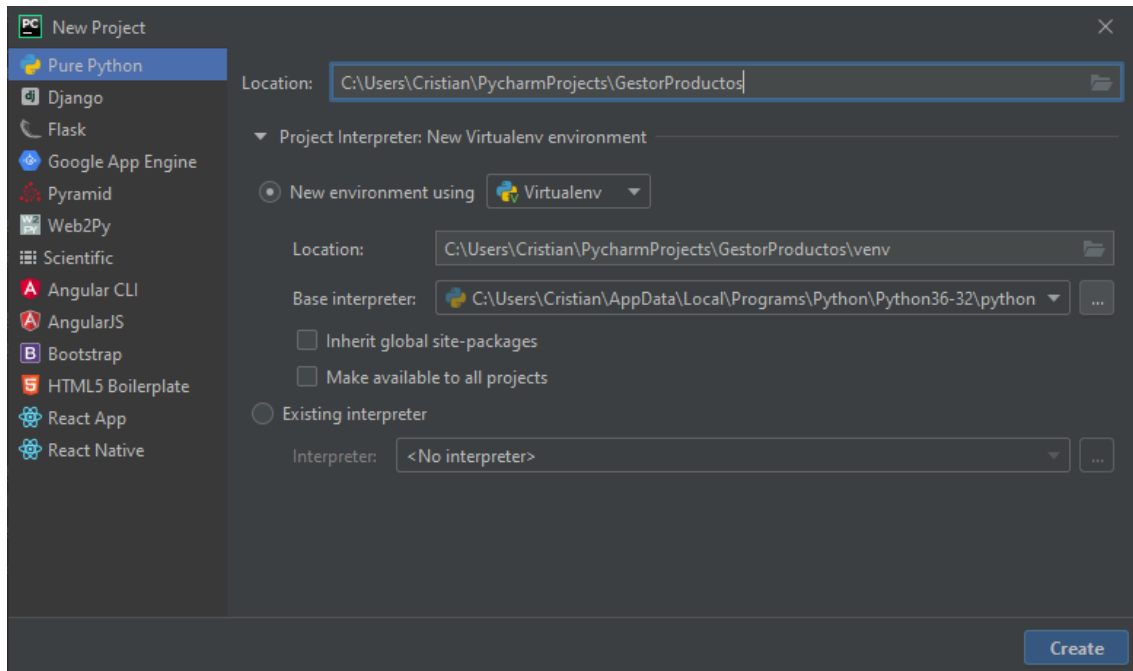
En un mundo ideal, se trabajaría en todos los proyectos con la misma versión de Python y con los mismos módulos o librerías. Pero la realidad es muy distinta, cada proyecto es totalmente diferente y utiliza versiones de Python o versiones de módulos o librerías diferentes. Por lo tanto, si se instalara en un sistema la versión de Python 3.6.2 por ejemplo y unos módulos o librerías determinados, todos los proyectos tendrían que utilizar esa versión de Python como ese listado de módulos y librerías instaladas. Esto, evidentemente, no es funcional ni práctico. Por eso, Python dispone de los entornos virtuales, lo que proporciona crear un entorno totalmente nuevo y limpio para cada proyecto. Pudiendo de esta forma, tener en un único sistema, en un único equipo, multitud de entornos virtuales para multitud de proyectos, y donde cada entorno virtual estará configurado de una manera. Ejemplo:

- Entorno virtual 1: Python 3.6.2 con el módulo SQLAlchemy (v2.5) y Pandas (v1.2)
- Entorno virtual 2: Python 3.1 con el módulo SQLAlchemy (v2.0) y Pandas (v1.2)
- Etc.

Esta es la forma en la que se trabaja profesionalmente, utilizando entornos virtuales para los proyectos. Por lo que se va a crear este proyecto siguiendo esta metodología.

1. Abrir el IDE de Python con el que se programará. En este caso, será Pycharm.
2. Crear un nuevo proyecto
 - File > New Project... >
 - Indicar ubicación y nombre del proyecto, en este caso, GestorProductos y en la ubicación por defecto, en la carpeta de proyectos de PyCharm
3. Seleccionar "New environment using > Virtualenv"

Virtualenv es la herramienta por defecto para crear entornos virtuales.



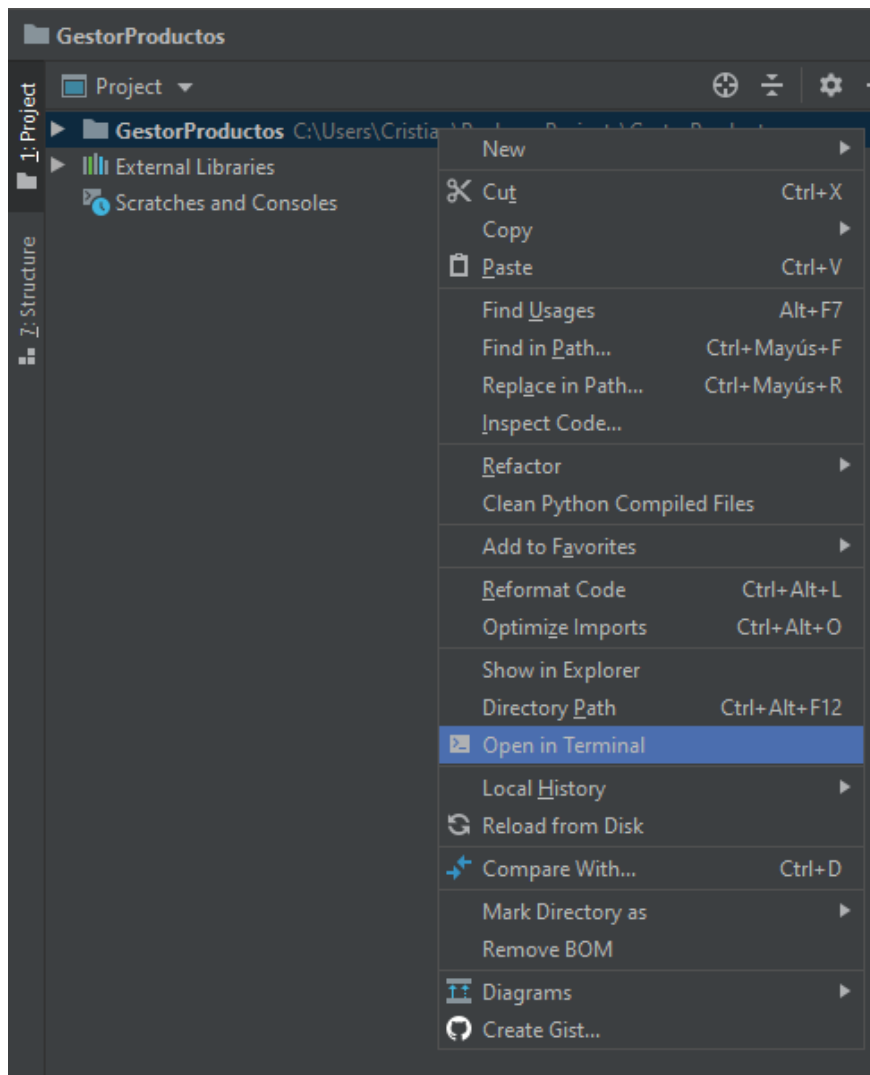
En este punto se tiene creado el proyecto, aunque vacío de momento.

4. Abrir un terminal del proyecto

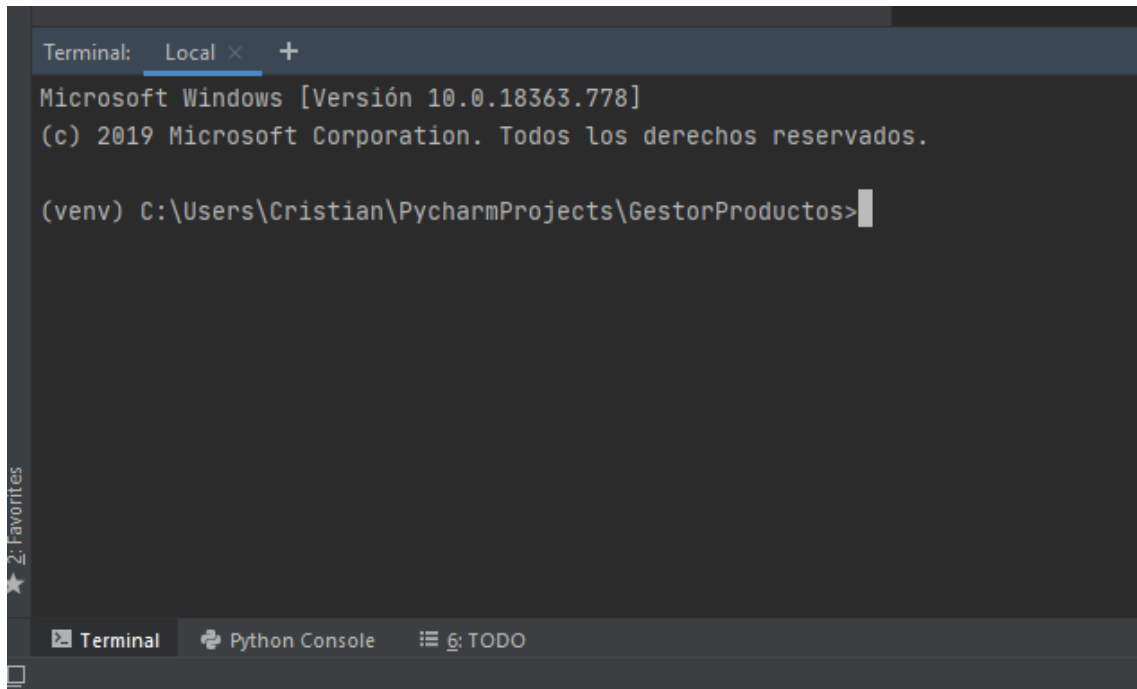
- Para ello se hará click derecho sobre el proyecto y se hará click en Open in Terminal

Nota. Para abrir un terminal en Visual Studio Code

- Presionar Ctrl + Shift + P
 - Esto abrirá un desplegable de operaciones.
- Seleccionar:
 - Terminal: *Create New Integrated Terminal*

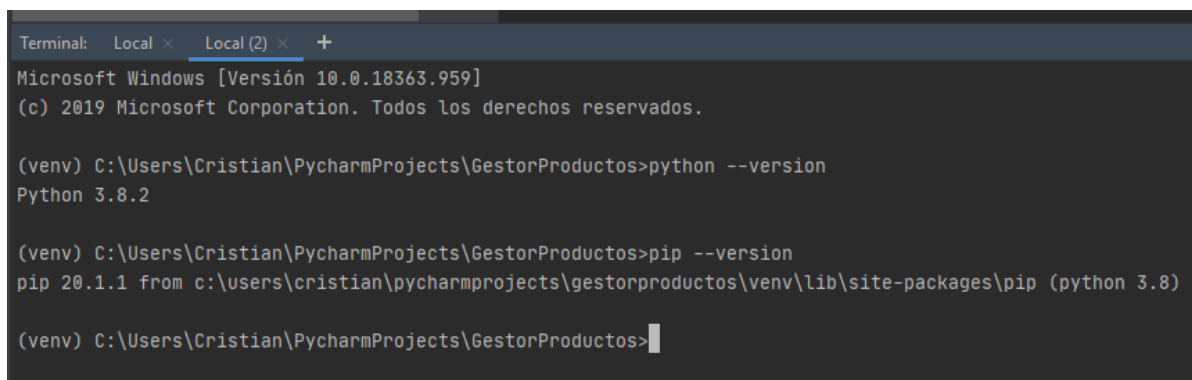


Esto abrirá un terminal (una consola) dentro del IDE y ya ubicado en el proyecto en cuestión.



```
Terminal: Local × +  
Microsoft Windows [Versión 10.0.18363.778]  
(c) 2019 Microsoft Corporation. Todos los derechos reservados.  
  
(venv) C:\Users\Cristian\PycharmProjects\GestorProductos>
```

5. Comprobar que se tiene acceso a Python desde la consola integrada del IDE:
- Se comprobará Python y pip (el instalador de módulos de Python que se necesitará más adelante).

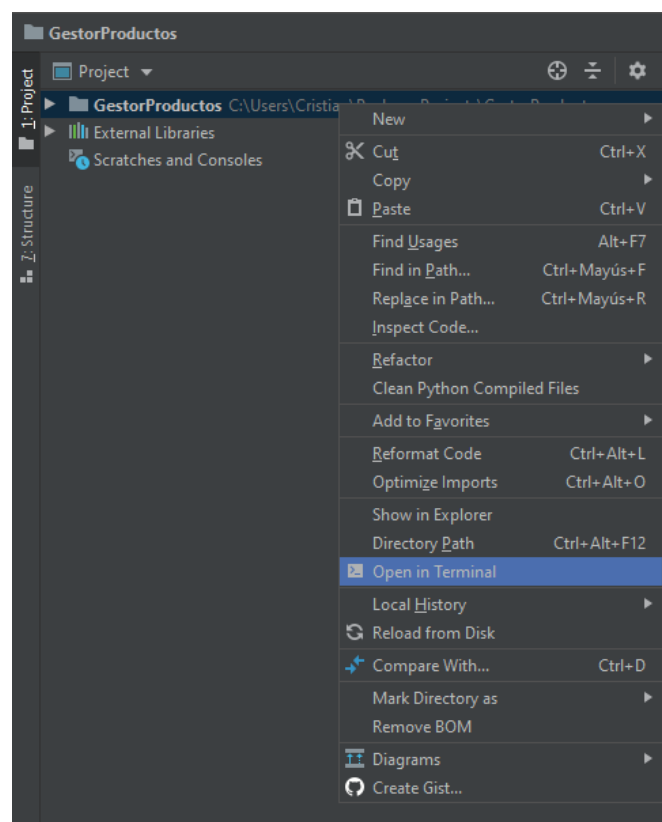


```
Terminal: Local × Local (2) × +  
Microsoft Windows [Versión 10.0.18363.959]  
(c) 2019 Microsoft Corporation. Todos los derechos reservados.  
  
(venv) C:\Users\Cristian\PycharmProjects\GestorProductos>python --version  
Python 3.8.2  
  
(venv) C:\Users\Cristian\PycharmProjects\GestorProductos>pip --version  
pip 20.1.1 from c:\users\cristian\pycharmprojects\gestorproductos\venv\lib\site-packages\pip (python 3.8)  
  
(venv) C:\Users\Cristian\PycharmProjects\GestorProductos>
```


3. Salir y entrar del entorno virtual

Si se cierra el IDE, en este caso Pycharm, también se cierra el entorno virtual en el que se está trabajando. Cuando se vuelve a abrir Pycharm se deberá volver a entrar al entorno virtual para seguir trabajando. Esto se realiza siguiendo los siguientes pasos:

1. Clic derecho sobre el directorio principal del proyecto.
2. Clic en Open in Terminal.



3. Si aparece un (venv) de virtual environment delante del Shell del proyecto, todo está correcto:

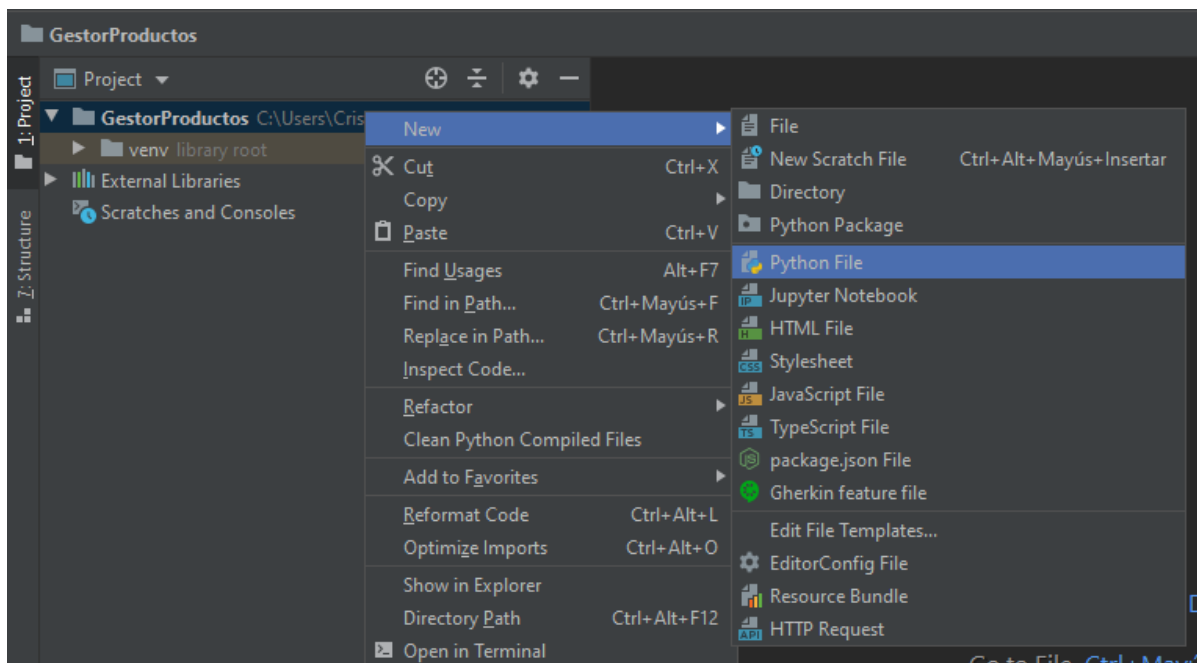
```
(venv) C:\Users\Cristian\PycharmProjects\GestorProductos>
```

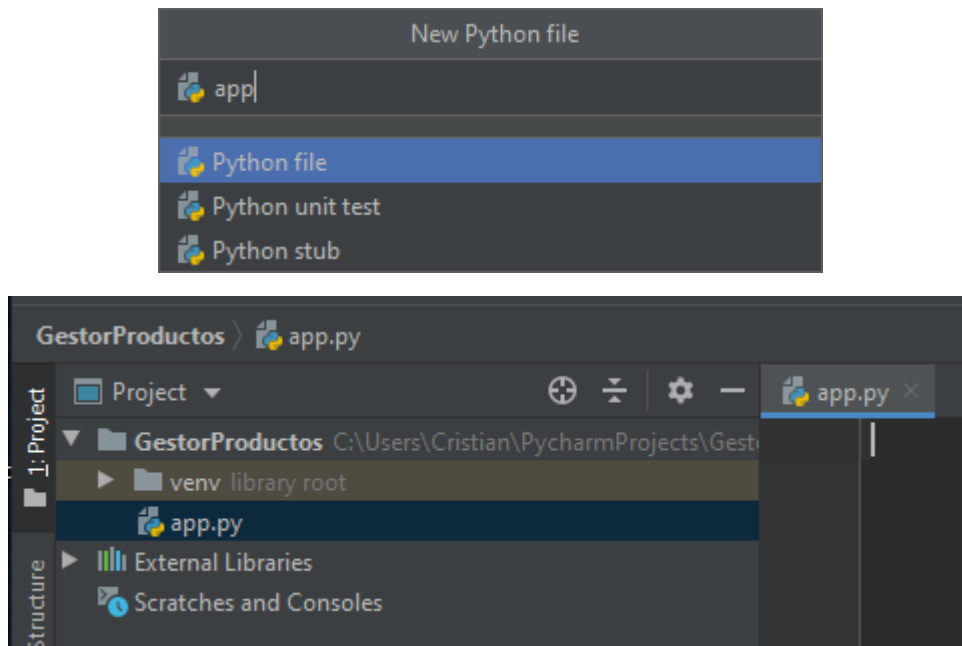
4. Creación del fichero Python principal y primera ventana

Aunque no existe ninguna restricción técnica a la hora de escoger el nombre de nuestros ficheros Python, hay ciertos estándares de la comunidad que nos indican los nombres estándar que se suelen utilizar, por ejemplo, en páginas web HTML, se suele poner index.html como fichero principal, o en hojas de estilo, se suele poner main.css. En Python también tenemos algunos nombres preferidos por la comunidad.

1. Creación del fichero principal de Python para este proyecto: app.py (nombre estándar para las aplicaciones de Tkinter).
 - Clic derecho sobre el directorio principal del proyecto.
 - Clic en New > Python File.
 - Se indica el nombre app y se hace doble clic izquierdo sobre Python file
 - Se crea en la raíz del proyecto el fichero app.py

Ver las capturas que se muestran a continuación para seguir los pasos.



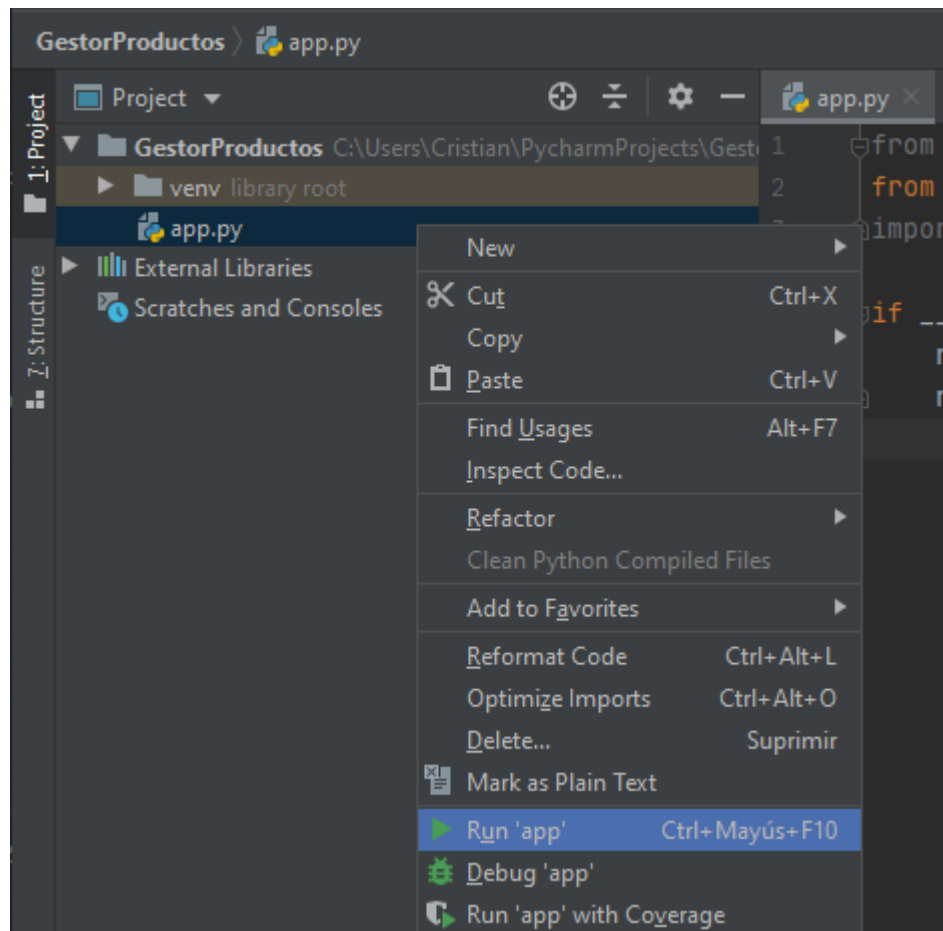


2. El primer objetivo, dado que esto es un proyecto para crear una aplicación de escritorio, es crear una ventana principal (ventana raíz o *root* como se llama en Tkinter), a continuación, se muestra el código mínimo para implementar esto (incluyendo el *import* a la librería de Tkinter y a SQLite que se utilizará más adelante):

```
from tkinter import ttk
from tkinter import *
import sqlite3

if __name__ == '__main__':
    root = Tk() # Instancia de la ventana principal
    root.mainloop() # Comenzamos el bucle de aplicacion, es como un while True
```

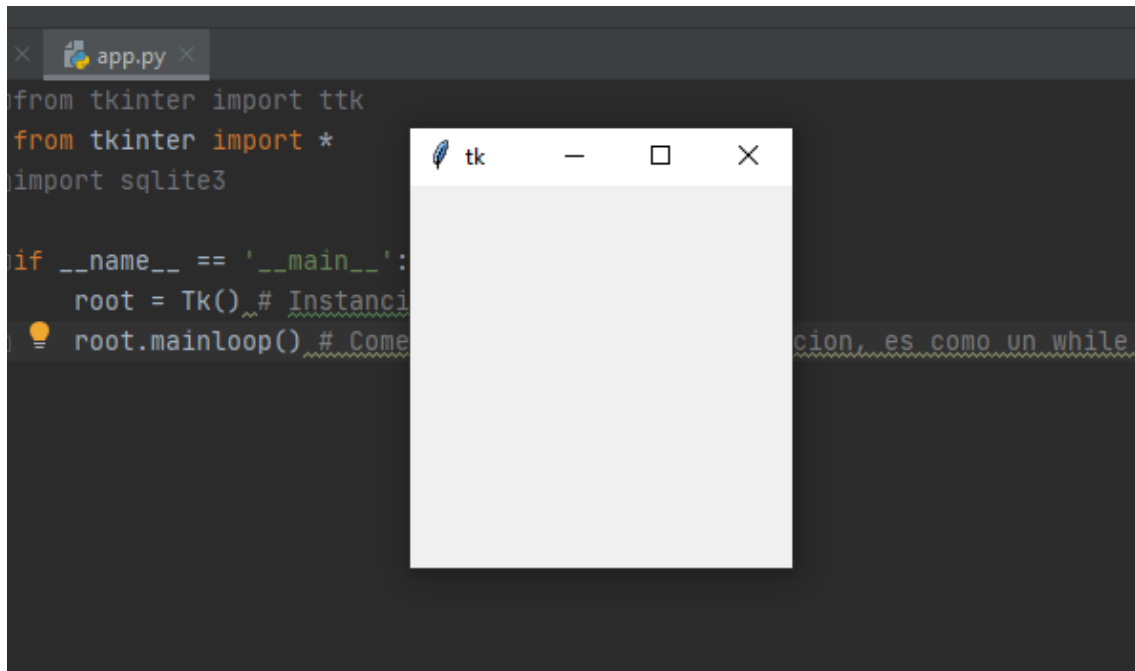
3. Ejecutar la aplicación (dos formas):
 - A través del botón de Run app de Pycharm



- A través de la consola del proyecto.

```
(venv) C:\Users\Cristian\PycharmProjects\GestorProductos>python app.py
```

4. El resultado será una ventana vacía de Tkinter:



5. Comenzar la implementación usando Clases y Objetos

1. Se podría programar todo en el *main* pero cuando la aplicación sea de un tamaño considerable, se verá que no es la mejor organización, por lo que se va a comenzar a implementar clases y objetos desde un inicio. Se creará una clase *Producto*, la cual recibirá como parámetro el control de la ventana gráfica de la aplicación y pasará a llamarla *ventana*.

```
class Producto:

    def __init__(self, root):
        self.ventana = root

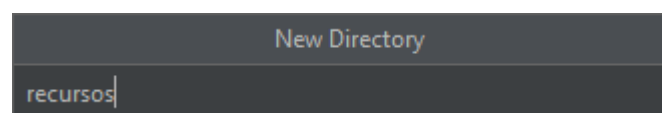
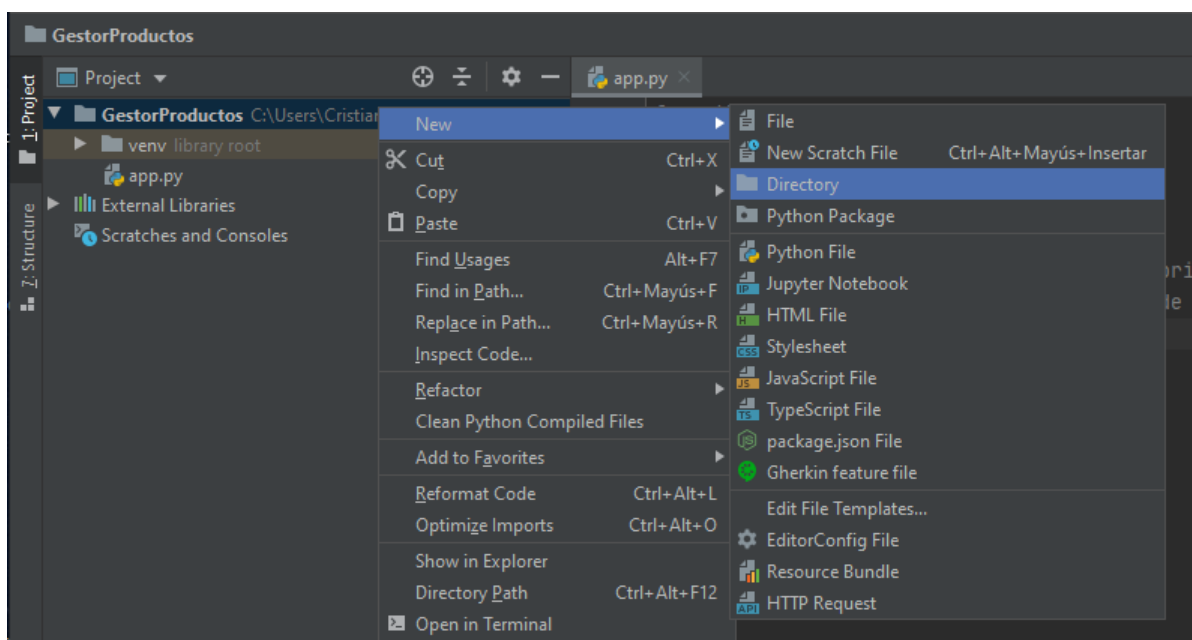
if __name__ == '__main__':
    root = Tk() # Instancia de la ventana principal
    app = Producto(root) # Se envia a la clase Producto el control sobre la
    ventana root
    root.mainloop() # Comenzamos el bucle de aplicacion, es como un while True
```

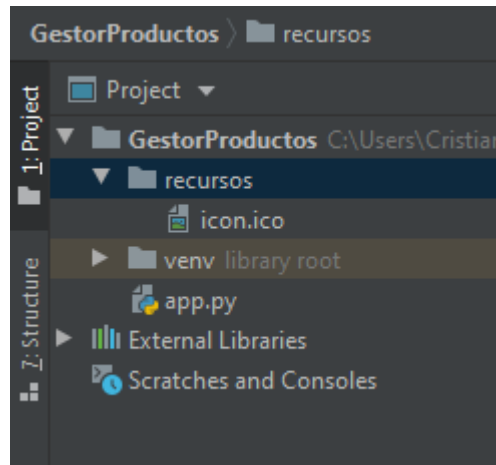
6. Configuración base de la ventana principal

1. Dentro del constructor de la clase Producto se tiene la variable que controla la ventana principal, llamada ventana. Pero adicionalmente se necesitan algunas cosas más:
 - Un nombre para la aplicación, que aparecerá en la barra de título de la aplicación (por defecto es tk).
 - Indicarle si se quiere que la ventana se redimensione o no.
 - Un icono (por defecto aparece el icono de Tkinter, la pluma).



2. Para poder añadir el icono (imagen con extensión .ico), se creará una carpeta en la raíz del proyecto que se llamará recursos. Y se copiará el icono llamado icon.ico que se proporciona con este proyecto a dicha carpeta.



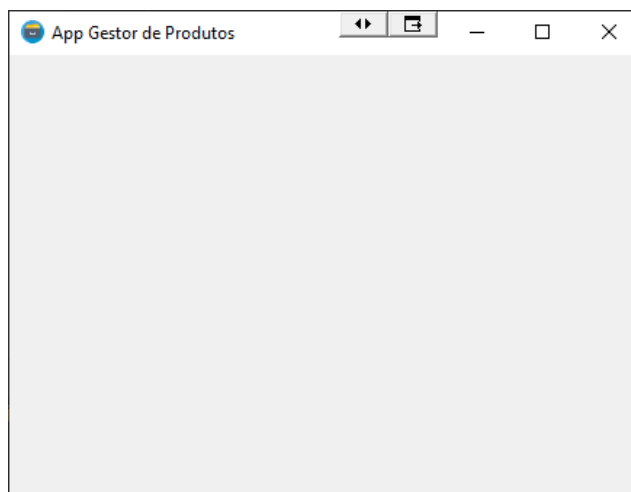


3. Se realizan las pertinentes modificaciones del código en el constructor de la clase Producto

```
class Producto:

    def __init__(self, root):
        self.ventana = root
        self.ventana.title("App Gestor de Productos") # Título de la ventana
        self.ventana.resizable(1,1) # Activar la redimension de la ventana. Para
desactivarla: (0,0)
        self.ventana.wm_iconbitmap('recursos/icon.ico')
```

4. Se prueba:





7. Estructura de los widgets (componentes gráficos) en ventana

Antes de comenzar a situar widgets (componentes gráficos) en la ventana, hay que entender cómo se estructura Tkinter.

Tkinter distribuye la ventana en una cuadrícula llamada *grid*

Columnas

Filas

A este *grid* se accede a través de su número de fila y columna.

Columnas

Filas

	F0 C0	F0 C1	F0 C2	
	F2 C0	F2 C1	F2 C2	

Y el objetivo del *grid* es lograr posicionar widgets en pantalla, en posiciones específicas. Los widgets son componentes gráficos, los cuales pueden ser botones, etiquetas de texto, cajones de texto para que el usuario pueda introducir texto, imágenes, sliders, etc.

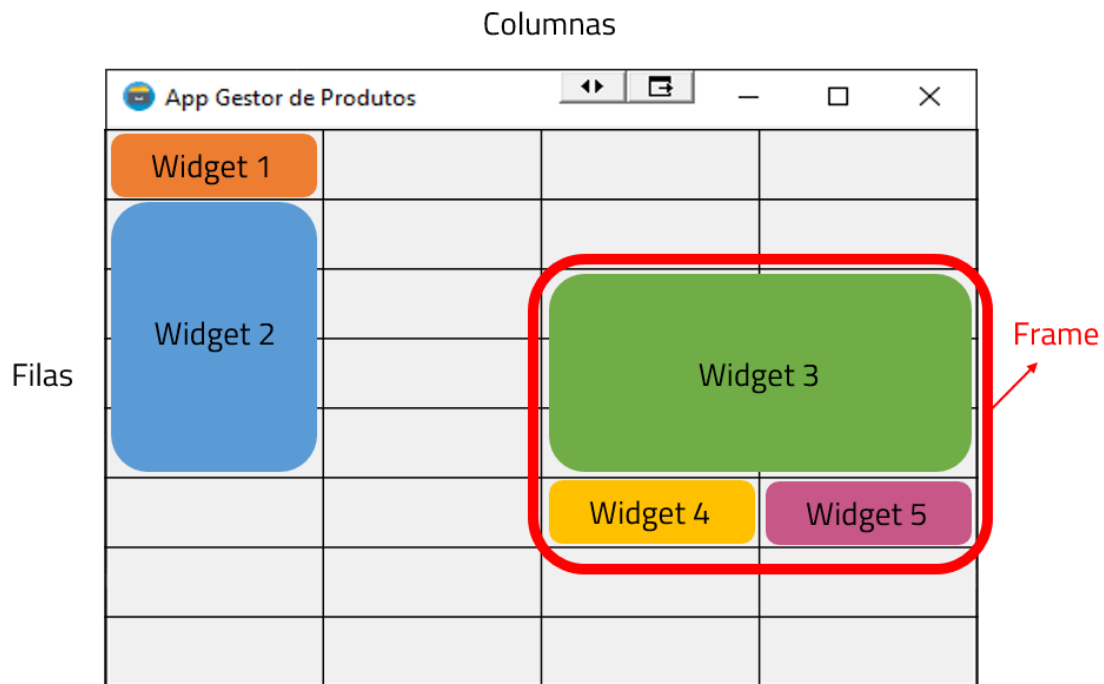
Columnas

Filas

	Widget 1			
	Widget 2			
			Widget 3	

Para posicionar widgets individuales, el *grid* por si sólo nos serviría, pero en muchas ocasiones se necesita crear agrupaciones de widgets, en estos casos, se crea otra estructura llamada *frame*.

Los *Frames* son marcos contenedores de otros widgets (componentes gráficos). Pueden tener tamaño propio y posicionarse en distintos lugares de otro contenedor (ya sea la raíz u otro marco):





8. Widgets utilizados en este proyecto

Tkinter tiene disponible una gran cantidad de componentes gráficos divididos en dos sub librerías diferentes.

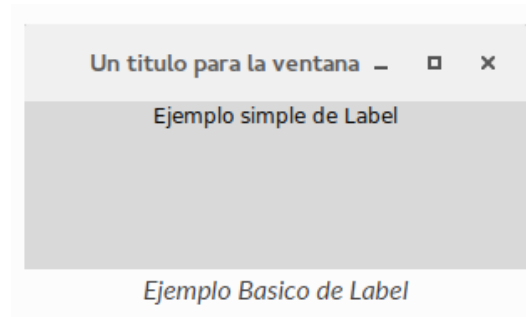
- Tk son los componentes más básicos y nativos (configuración muy sencilla).
- Ttk son componentes más complejos, con una configuración más complicada, pero con un diseño más cuidado.

tk	ttk
Button	Button
Canvas	Checkbutton
Checkbutton	Combobox
Entry	Entry
Frame	Frame
Label	Label
LabelFrame	LabeledScale
Listbox	Labelframe
Menu	Menubutton
Menubutton	Notebook
Message	OptionMenu
OptionMenu	Panedwindow
PanedWindow	Progressbar
Radiobutton	Radiobutton
Scale	Scale
Scrollbar	Scrollbar
Spinbox	Separator
Text	Sizegrip
	Treeview

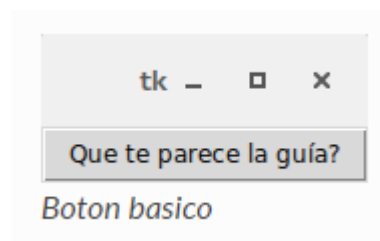
En la bibliografía se incluye un enlace a la guía no oficial de Tkinter en español, concretamente a la sección de los widgets.

A continuación, se listan los *widgets* que se utilizarán en este proyecto, con sus correspondientes nombres técnicos dentro de Tkinter:

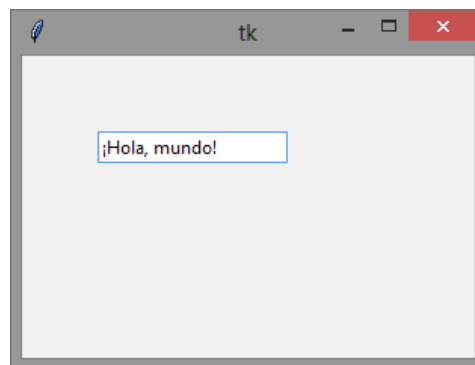
- Label (Etiqueta). Etiqueta de texto.



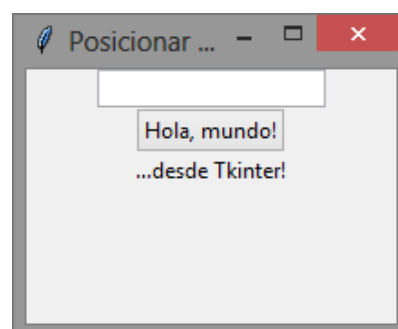
- Button (Botón).



- Entry (Cuadro de texto). Sirve para que el usuario pueda escribir en su interior.



Ejemplo de los 3 *widgets* anteriores juntos:



9. Posicionamiento de los elementos en Tkinter

En Tkinter se tienen 3 formas de insertar *widgets* o *frames* en la ventana:

1. Directamente en la grid (la cuadrícula de la ventana).
2. Posicionamiento absoluto.
3. Posicionamiento relativo.

A continuación, se verán en detalle:

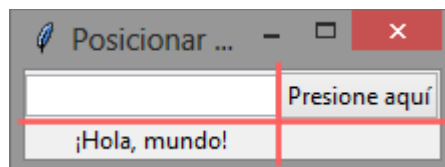
1. Grid (método *grid*)

Consiste en dividir conceptualmente la ventana principal en filas (*rows*) y columnas (*columns*), formando celdas en donde se ubican los elementos.

Tomemos como ejemplo una etiqueta *label* que contiene el texto "Hola mundo" y que se quiere ubicar en la fila 1, columna 0

```
self.label = ttk.Label(self, text="¡Hola, mundo!")  
self.label.grid(row=1, column=0)
```

Como se ve, se utiliza el método *grid()*, el cual tiene dos parámetros, la fila y la columna.



El método *grid* acepta, al igual que *pack()*, los argumentos *padx*, *pady*, *ipadx* e *ipady* para establecer márgenes (se verán en el punto 3, posicionamiento relativo).

2. Posicionamiento absoluto (método *place*)

El método *place()* permite ubicar elementos indicando su posición (X e Y) respecto de un elemento padre. En general casi todas las librerías gráficas proveen una opción de este tipo, ya que es la más intuitiva.

Tomemos como ejemplo un botón que se quiera ubicar en la posición (60, 40)

```
self.button = ttk.Button(self, text="Hola, mundo!")
```

```
self.button.place(x=60, y=40)
```

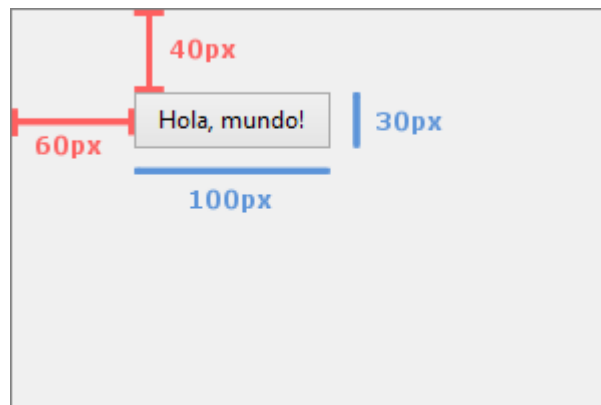
Ya que el origen de coordenadas (es decir, la posición (0, 0)) es la esquina superior izquierda, esto quiere decir que entre el borde izquierdo de la ventana y nuestro botón habrá una distancia de 60 píxeles y entre el borde superior de la ventana y el botón habrá 40 píxeles.



Es posible indicar el tamaño de cualquier otro elemento de Tkinter utilizando los parámetros `width` y `height`, que indican el ancho y el alto en píxeles.

```
self.button.place(x=60, y=40, width=100, height=30)
```

La siguiente imagen ilustra cómo influyen los cuatro argumentos (`x`, `y`, `width`, `height`) en la posición y el tamaño del *widget*.



3. Posicionamiento relativo (método *pack*)



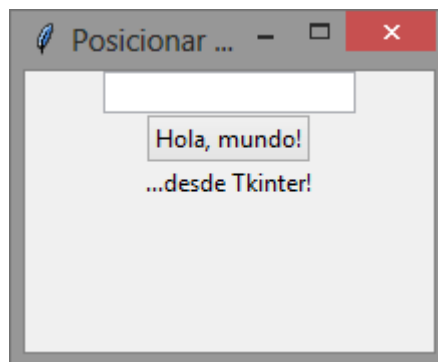
Este método es el más sencillo de los tres. En lugar de especificar las coordenadas de un elemento, simplemente le decimos que debe ir arriba, abajo, a la izquierda o a la derecha respecto de algún otro *widget* o bien de la ventana principal.

Tomemos como ejemplo, el diseño de un *Entry* (caja de texto), un botón y una *label* (etiqueta de texto). Para ubicar los elementos utilizaremos el método `pack()`, y si lo utilizamos sin argumentos, por defecto, colocará los elementos unos encima de los otros.

```
self.entry = ttk.Entry(self)
self.entry.pack()
```

```
self.button = ttk.Button(self, text="Hola, mundo!")
self.button.pack()
```

```
self.label = ttk.Label(self, text="...desde Tkinter!")
self.label.pack()
```

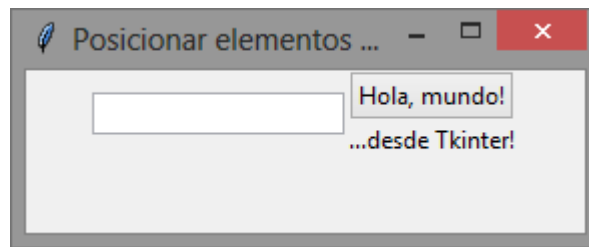


Pero `pack()` es mucho más potente que esto, y nos proporciona la capacidad de colocar los elementos donde se quiera, gracias a la propiedad `side`, la cual controla la posición relativa de los elementos:

- `Tk.TOP` (por defecto)
- `Tk.BOTTOM`
- `Tk.LEFT`
- `Tk.RIGHT`

De este modo, si indicamos que la caja de texto debe ir ubicada a la izquierda, los otros dos *widgets* se seguirán manteniendo uno arriba del otro.

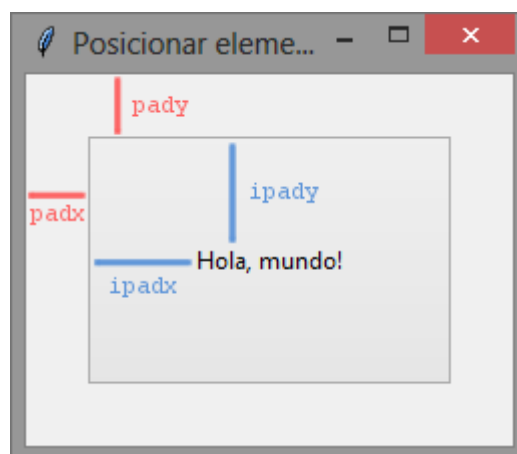
```
self.entry = ttk.Entry(self)
self.entry.pack(side=tk.LEFT)
```



Otras propiedades interesantes son *after* y *before* para indicar que un *widget* debe ir antes o después que otro.

Y otras propiedades son *padx*, *ipadx*, *pady* y *ipady* que especifican (en píxeles) los márgenes externos e internos de un elemento. Por ejemplo, en el siguiente código habrá un espacio de 30 píxeles entre el botón y la ventana (margen externo), pero un espacio de 50 píxeles entre el borde del botón y el texto del mismo (margen interno).

```
self.button = ttk.Button(self, text="Hola, mundo!")
self.button.pack(padx=30, pady=30, ipadx=50, ipady=50)
```

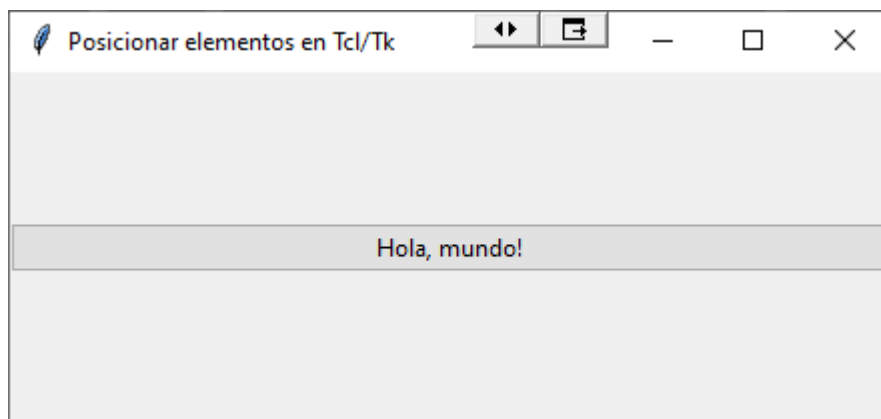




Por último, es posible especificar qué elementos deben expandirse o contraerse a medida que el tamaño de la ventana cambia, y en qué sentido deben hacerlo (vertical u horizontal), vía las propiedades `expand` y `fill`.

```
self.button = ttk.Button(self, text="Hola, mundo!")  
self.button.pack(expand=True, fill=tk.X)  
self.pack(expand=True, fill=tk.BOTH)
```

En el ejemplo, le indicamos al elemento padre (`self`) que ocupe todo el tamaño posible (`expand=True`) y que lo haga en ambas direcciones (`fill=tk.BOTH`). El botón, por otra parte, únicamente ajustará su tamaño horizontal (`fill=tk.X`).





10. Comencemos con la interfaz gráfica. Añadir Producto

1. Creación de un frame principal que englobe todos los widgets referentes a la funcionalidad de añadir producto. Ubicados en el constructor de la clase Producto se añade lo siguiente:

```
# Creacion del contenedor Frame principal
frame = LabelFrame(self.ventana, text = "Registrar un nuevo Producto")
frame.grid(row = 0, column = 0, columnspan = 3, pady = 20)
```

LabelFrame es un *widget* hijo de *Frame*. Se trata de un contenedor de *widgets* al igual que su padre. La diferencia radica en que dibuja un borde en torno a su tamaño.

Este *frame* se inicia en la posición 0,0 de la ventana, y con *columnspan* se indica cuantas columnas del *grid* va a utilizar. También se le incluye un poco de margen en el eje y con la instrucción *pady*.

2. Creación de la *Label* y el *Entry* de Nombre

Seguimos en el constructor de la clase Producto y se añade lo siguiente:

```
# Label Nombre
self.etiqueta_nombre = Label(frame, text="Nombre: ") # Etiqueta de texto ubicada
en el frame
self.etiqueta_nombre.grid(row=1, column=0) # Posicionamiento a traves de grid
# Entry Nombre (caja de texto que recibira el nombre)
self.nombre = Entry(frame) # Caja de texto (input de texto) ubicada en el frame
self.nombre.focus() # Para que el foco del raton vaya a este Entry al inicio
self.nombre.grid(row=1, column=1)
```

Como se observa, se crea la Label de nombre, indicando que va a existir dentro del *frame*. Y se coloca mediante el método *grid()* en la fila 1 y columna 0. A continuación se crea el Entry de nombre, el cajón de texto donde los usuarios escribirán el nombre. Se le indica que



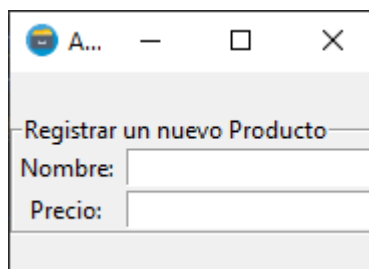
también existe dentro del *frame* y se ubica también a través del método `grid()` en la fila 1 y columna 1 (es decir, a la derecha de su *Label*).

Por último también se utiliza el método `focus()` para centrar el foco del ratón en el Entry de Nombre, es decir, que cuando se ejecute la aplicación, el Entry de Nombre sea directamente accesible.

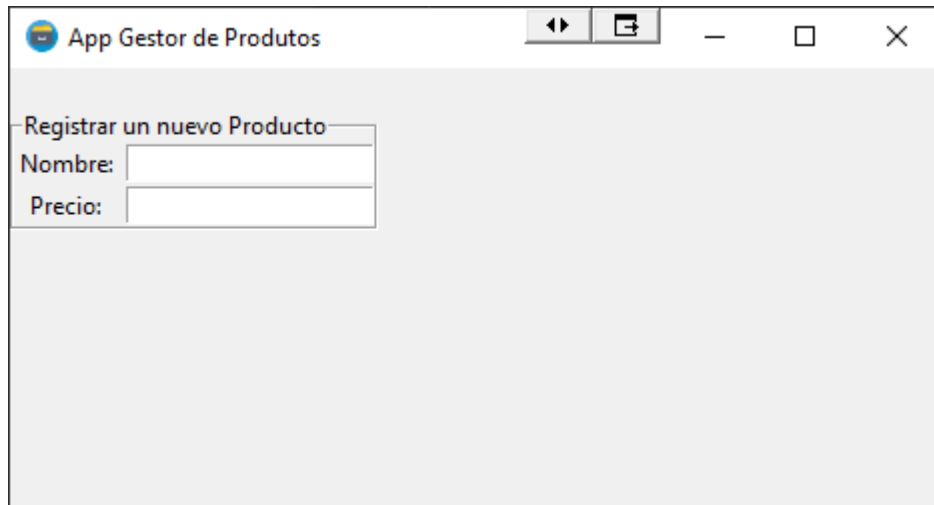
3. Creación de la Label y el Entry de Precio

```
# Label Precio
self.etiqueta_precio = Label(frame, text="Precio: ") # Etiqueta de texto ubicada
en el frame
self.etiqueta_precio.grid(row=2, column=0)
# Entry Precio (caja de texto que recibira el precio)
self.precio = Entry(frame) # Caja de texto (input de texto) ubicada en el frame
self.precio.grid(row=2, column=1)
```

4. Comprobación de lo implementado



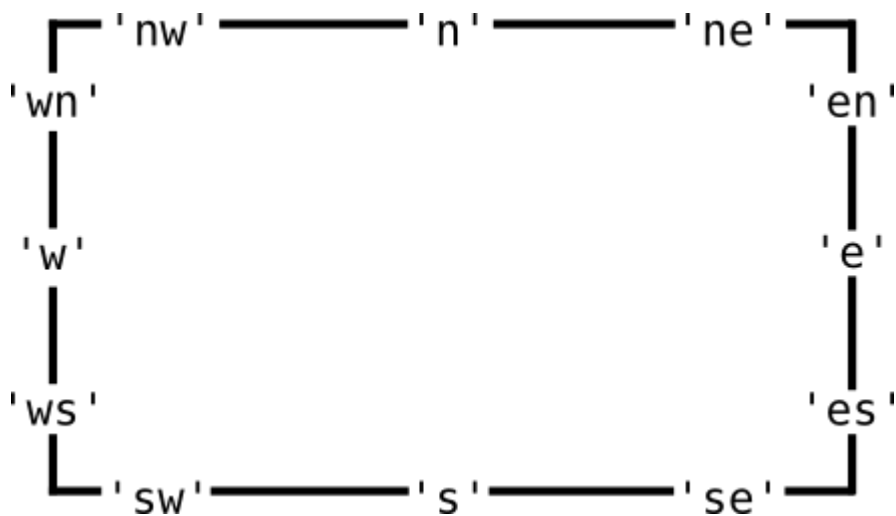
Si en las propiedades del constructor le indicamos que la ventana pudiera ser redimensionada (`self.ventana.resizable(1,1)`) se puede hacer la ventana más grande para ver todos los detalles de la implementación, como márgenes, etc.



5. Creación del botón de Guardar Producto

```
# Boton Añadir Producto
self.boton_aniadir = ttk.Button(frame, text = "Guardar Producto")
self.boton_aniadir.grid(row = 3, columnspan = 2, sticky = W + E)
```

Para este caso, se ha evitado utilizar la Ñ en el código para evitar problemas sintácticos. Este código crea un botón con el texto Guardar Producto, dentro del *frame*, ubicado en la fila 3 y con el atributo sticky se le dice cuanto se quiere que ocupe. En este caso se le dice que ocupe todo el ancho, desde el oeste (W) hasta el este (E). Estas etiquetas de posicionamiento pertenecen al *frame* y a continuación se pueden ver todas las que hay disponibles.





6. Comprobación de lo implementado

A screenshot of a Tkinter window titled "Registrar un nuevo Producto". The window has a standard macOS-style title bar with a red, yellow, and green icon on the left, and minimize, maximize, and close buttons on the right. The main content area contains two text input fields: "Nombre:" and "Precio:". Below these fields is a button labeled "Guardar Producto". The window is centered on the screen.

11. Interfaz gráfica. Tabla de Productos

En este punto, se tiene un formulario para que el usuario pueda guardar nuevos productos. Pero, desde la propia ventana de la aplicación, el usuario tendrá que ver los productos existentes. Para ello se va a crear una tabla (*widget Treeview* de la sub librería *ttk*).

1. Creación de la tabla de Productos. Se creará un estilo propio para la tabla, modificando el texto de las cabeceras, haciendo que sean más grandes y estén en negrita, eliminando los bordes de la tabla, etc. Las fuentes y tamaños se pueden cambiar libremente.

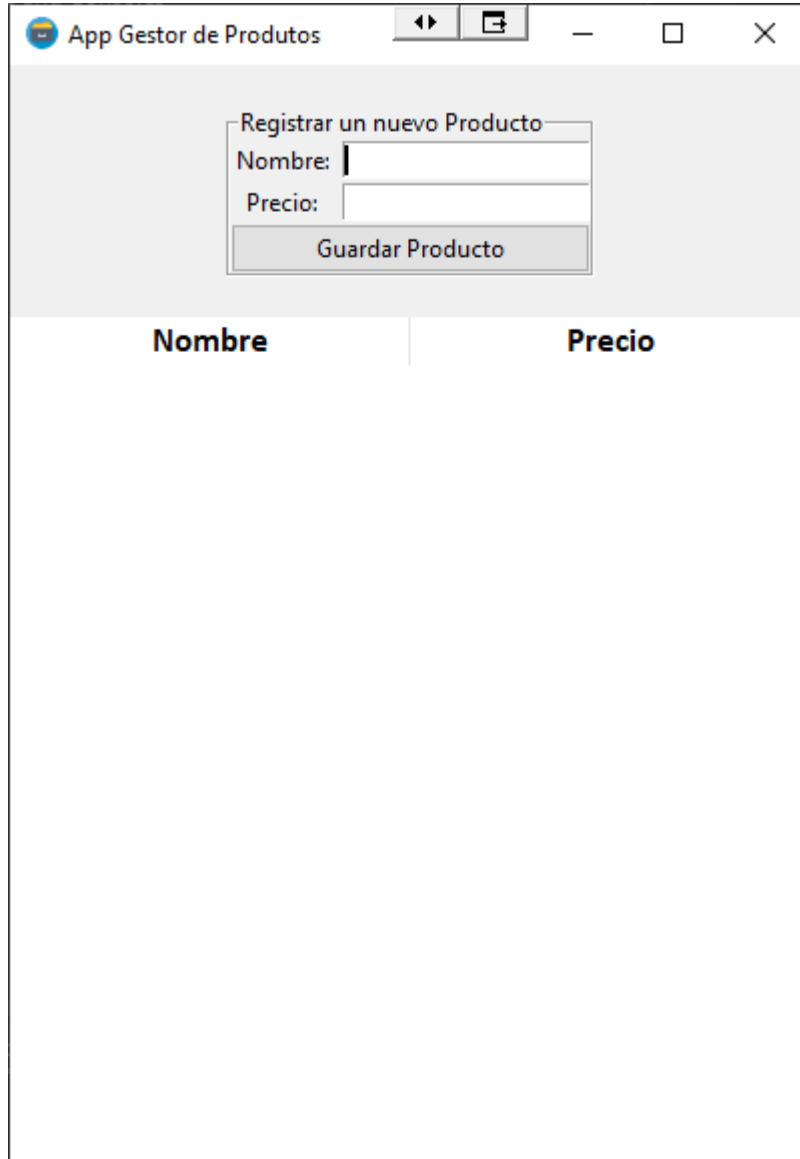
Continuamos en el constructor de la clase *Producto*:

```
# Tabla de Productos
# Estilo personalizado para la tabla
style = ttk.Style()
style.configure("mystyle.Treeview", highlightthickness=0, bd=0, font=('Calibri',
11)) # Se modifica la fuente de la tabla
style.configure("mystyle.Treeview.Heading", font=('Calibri', 13, 'bold')) # Se
modifica la fuente de las cabeceras
style.layout("mystyle.Treeview", [('mystyle.Treeview.treearea', {'sticky':
'nswe'})]) # Eliminamos los bordes

# Estructura de la tabla
self.tabla = ttk.Treeview(height = 20, columns = 2, style="mystyle.Treeview")
self.tabla.grid(row = 4, column = 0, columnspan = 2)
self.tabla.heading('#0', text = 'Nombre', anchor = CENTER) # Encabezado 0
self.tabla.heading('#1', text='Precio', anchor = CENTER) # Encabezado 1
```

En la parte de construcción de la estructura se le indica que tenga una altura de 20 filas y un ancho de 2 columnas. Esto se puede variar al gusto. Y se ubica con el método `grid()` en la fila 4 (falta ubicar un elemento en la fila 3 más adelante). Por último, se añaden los encabezados de la tabla.

2. Comprobación de lo implementado



App Gestor de Productos

Registrar un nuevo Producto

Nombre:

Precio:

Guardar Producto

Nombre	Precio
--------	--------

Ahora sería el momento de cargar de contenido la tabla con los productos. Para ello se necesitan datos, datos almacenados en una base de datos.

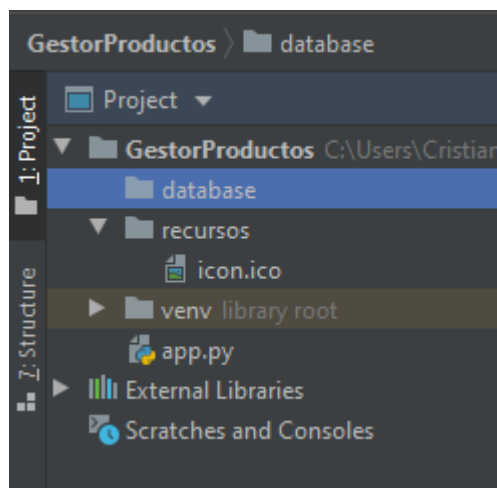
12. Creación de la base de datos

En este proyecto se utilizará una base de datos SQLite y el gestor DB Browser (for SQLite). Queda fuera del ámbito de este proyecto la instalación de ambos, por lo tanto, si en este punto no se tienen instalados en el ordenador, habrá que hacerlo de inmediato para poder continuar. Los enlaces a las páginas de descargas se encuentran en la bibliografía de este proyecto.

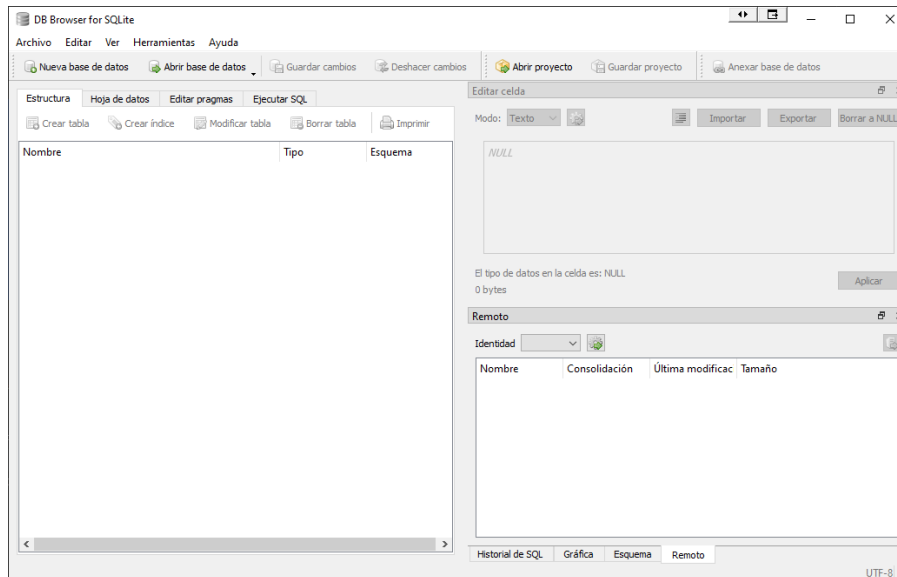
La creación y configuración de la base de datos que se detalla a continuación se podría hacer a través de código, pero para variar respecto a otros proyectos, se va a utilizar el programa DB Browser para llevarla a cabo.

1. Creación de la carpeta donde se almacenará la base de datos

Al igual que se hizo con la carpeta de recursos, se va a crear una carpeta llamada database



2. Se ejecutará DB Browser (for SQLite)



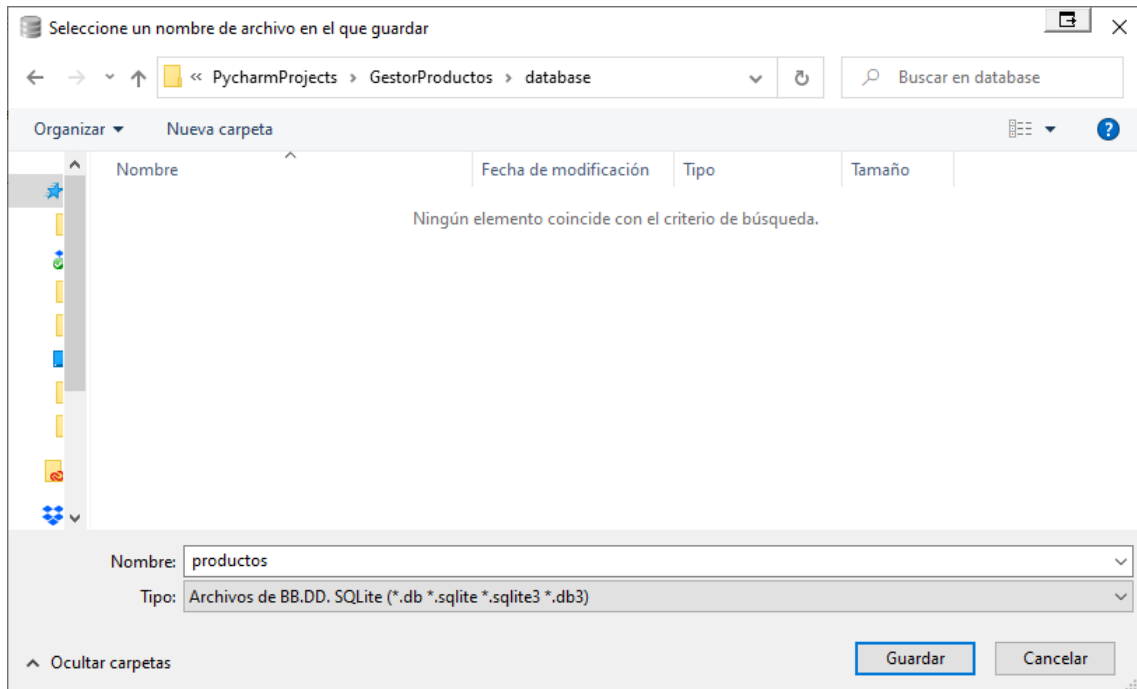
3. Se irá a Nueva base de datos en la parte superior izquierda.

Se abrirá una ventana emergente para indicar al programa donde se quiere crear la base de datos. Habrá que dirigirse a la carpeta que se acaba de crear PycharmProjects\GestorProductos\database.

En nuestro caso concreto, la ruta absoluta es:

C:\Users\Cristian\PycharmProjects\GestorProductos\database pero esta ruta dependerá del nombre del usuario del sistema y de la localización del *workspace* del IDE de desarrollo que se utilice.

Cuando la ubicación sea correcta, se le dará el nombre productos a la base de datos y se pulsará en Guardar.



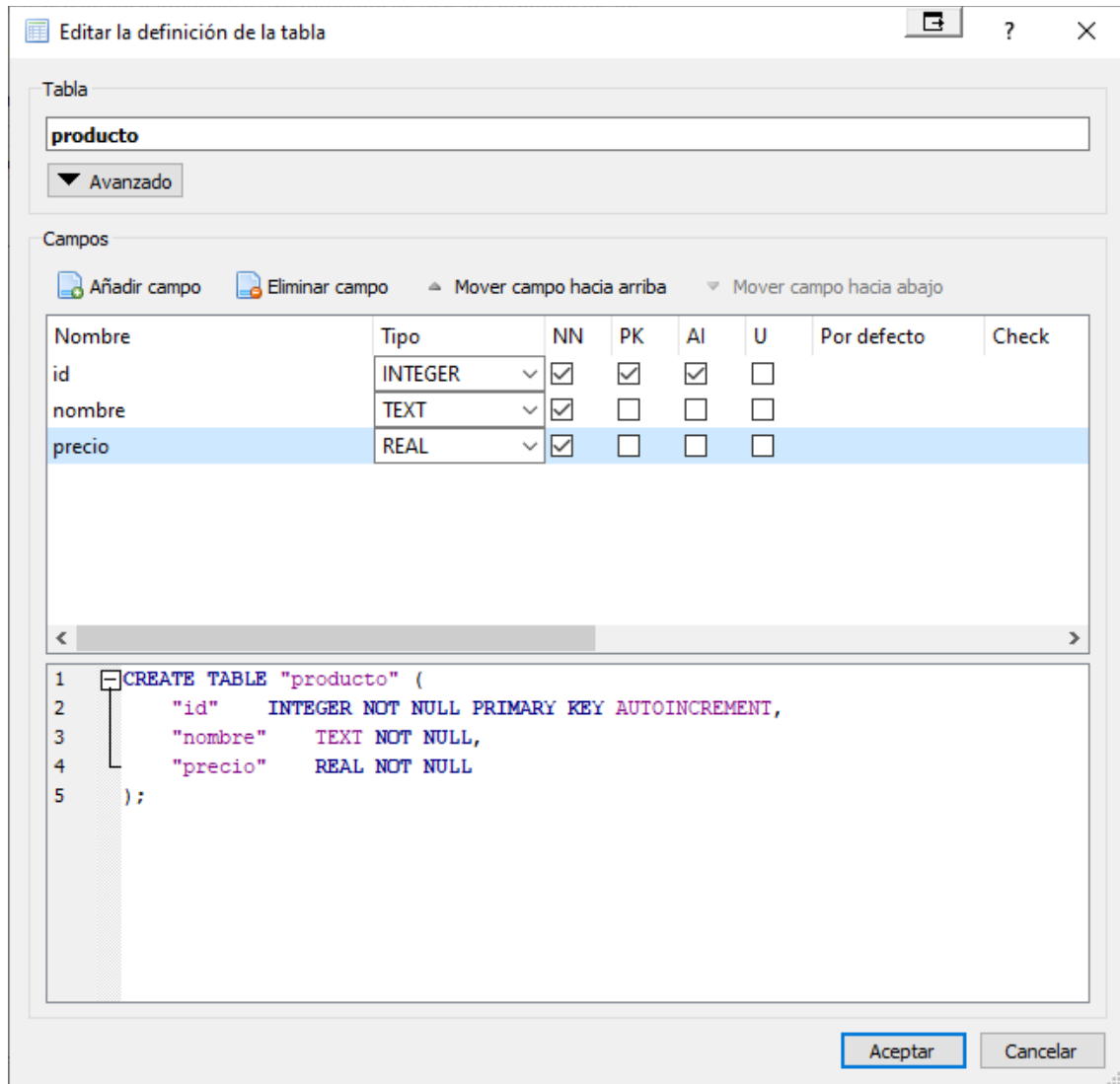
4. A continuación, se abre una ventana donde directamente DB Browser pregunta al usuario si desea crear una tabla para su base de datos. Vamos a crear la tabla producto. En bases de datos, hay un convenio que es nombrar los nombres de las bases de datos en plural y los nombres de las tablas en singular. Por eso esta base de datos se llama productos y la tabla producto.

Esta tabla producto tendrá 3 campos (para añadir los campos hay que pulsar en Añadir campo) con los siguientes atributos:

- id
 - Tipo: INTEGER (número entero)
 - NN: Not null (no puede ser nulo, no puede dejarse este campo vacío)
 - PK: Primary Key (clave primaria, campo único que no permitirá repetidos y servirá para identificar inequívocamente a cada fila de datos)
 - AI: AutoIncrement (número autoincremental). El gestor de la base de datos se encarga de generar automáticamente este número 1, 2, 3, ... sin repetidos.
- nombre
 - Tipo: TEXT

- NN: Not null (no puede ser nulo, no puede dejarse este campo vacío)
- precio
 - Tipo: REAL (número real, con decimales)
 - NN: Not null (no puede ser nulo, no puede dejarse este campo vacío)

El resultado de la configuración deberá ser el siguiente:



Editar la definición de la tabla

Tabla

producto

Avanzado

Campos

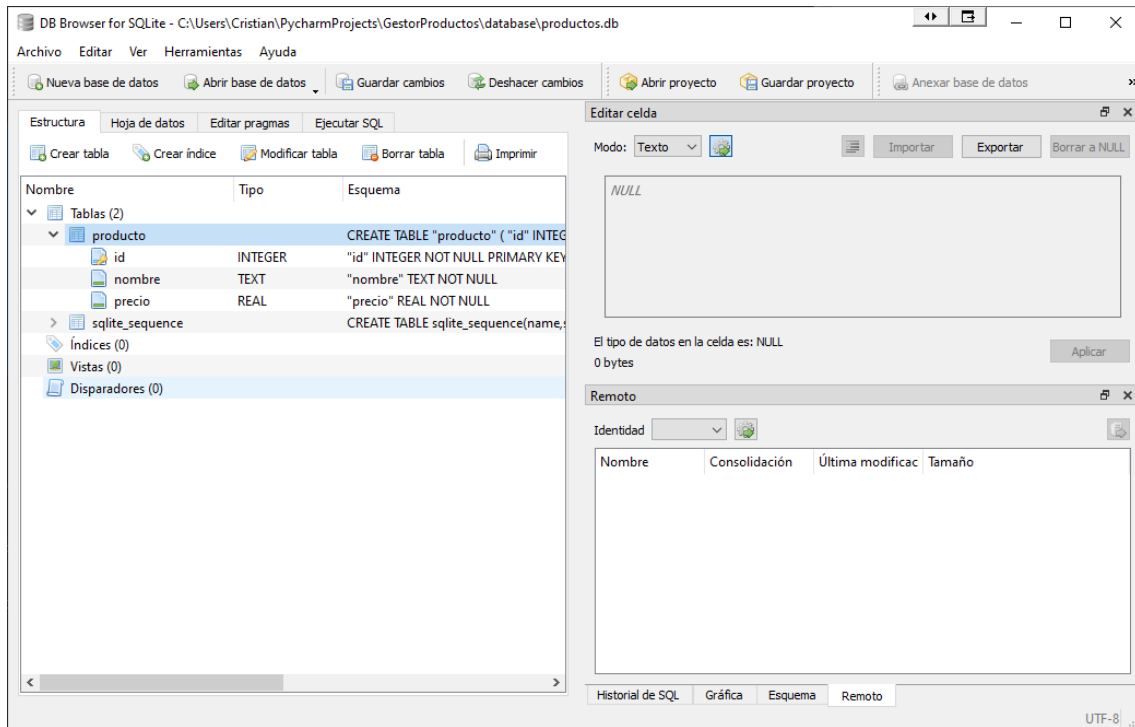
Añadir campo Eliminar campo Mover campo hacia arriba Mover campo hacia abajo

Nombre	Tipo	NN	PK	AI	U	Por defecto	Check
id	INTEGER	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>		
nombre	TEXT	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>		
precio	REAL	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>		

```
1 CREATE TABLE "producto" (  
2     "id"    INTEGER NOT NULL PRIMARY KEY AUTOINCREMENT,  
3     "nombre" TEXT NOT NULL,  
4     "precio" REAL NOT NULL  
5 );
```

Aceptar Cancelar

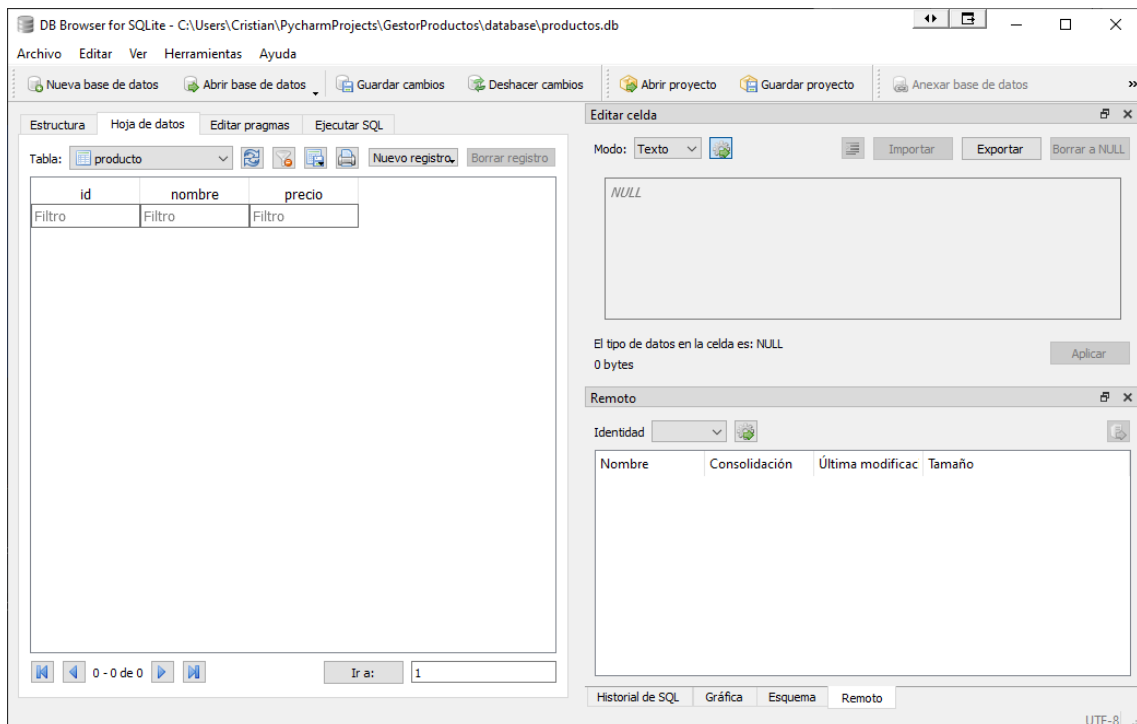
Al finalizar la configuración se pulsará en Aceptar para crear la tabla. Se volverá a la ventana principal de DB Browser donde se podrá ver la tabla recién creada.



13. Inserción de datos de prueba en la base de datos

Vamos a insertar unos registros (unas filas) de datos de muestra para poder acceder a ellos desde la aplicación, antes de implementar la funcionalidad de guardar nuevos productos.

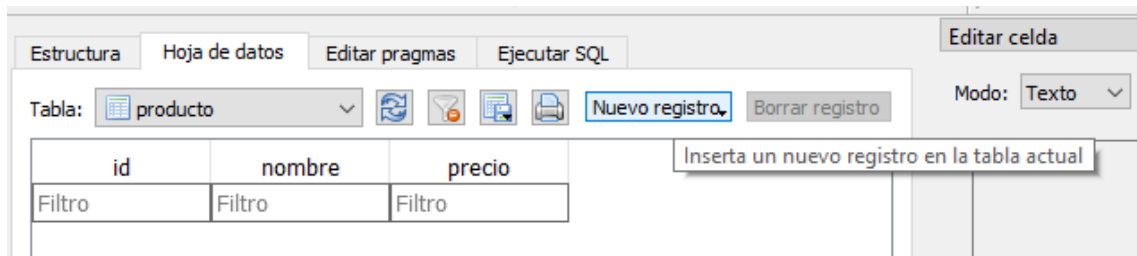
1. Para ello se irá a la pestaña Hoja de datos de DB Browser



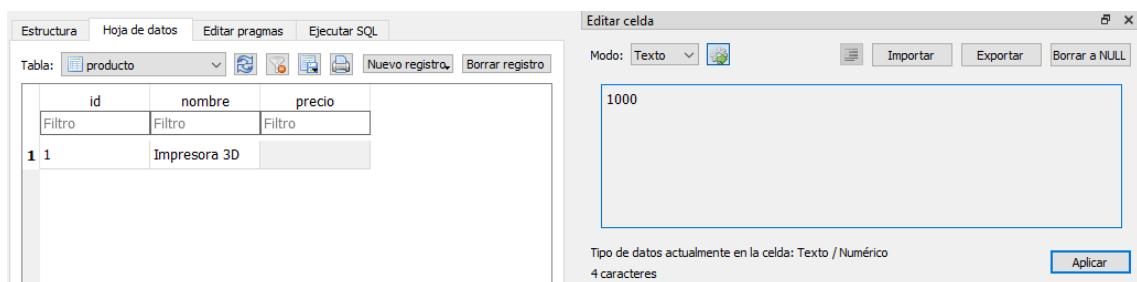
2. Se ingresarán los siguientes registros. Hay que tener en cuenta que el id del producto no hay que introducirlo manualmente ya que se configuro como autoincremental, por lo tanto, se encargará el gestor de añadirlo.
 - id 1
 - nombre: Impresora 3D
 - precio: 1000
 - id 2
 - nombre: Arduino
 - precio: 20
 - id 3
 - nombre: Raspberry Pi 4

o precio: 50

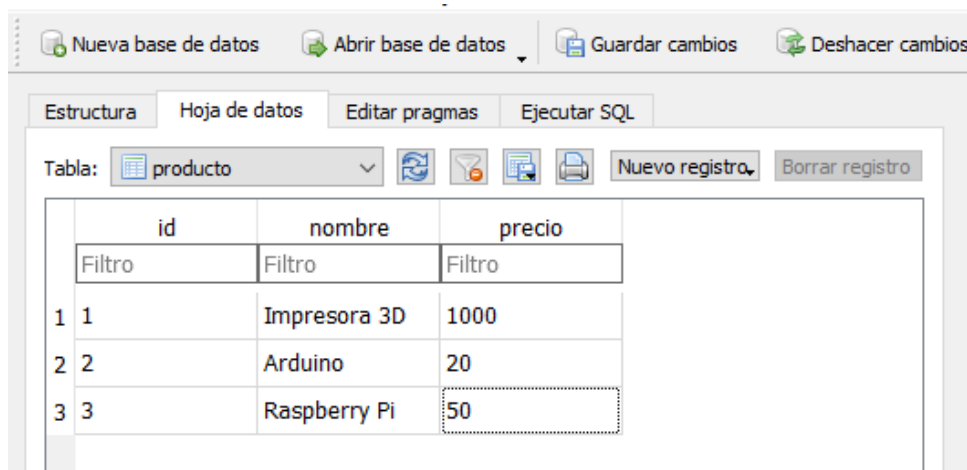
3. Para crear los registros se pulsará en Nuevo registro



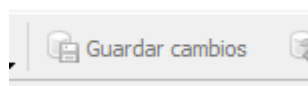
4. Tras pulsar en Nuevo registro se comprueba que el id lo inserta automáticamente. Para insertar los datos que se desean únicamente hay que hacer click sobre el nombre o el precio y rellenar con los datos en la pantalla de la derecha. Cuando se termine, se pulsa en Aplicar



5. Cuando se haya finalizado de insertar los registros se debe pulsar en Guardar cambios



Cuando los datos este guardados, el botón de Guardar cambios se desactivará:

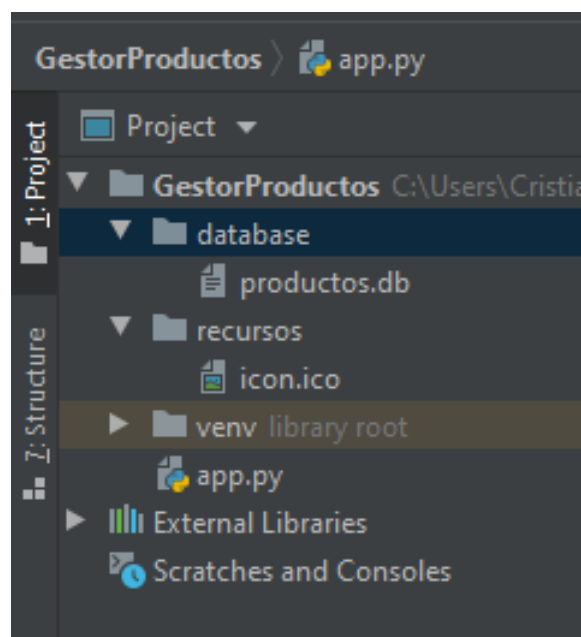




14. Implementar la conexión a la base de datos desde Python

En este punto, se tiene creada la base de datos con una tabla y varios registros insertados. Se va a programar la conexión a la base de datos para poder acceder a su contenido.

1. Comprobar que desde Pycharm se vea la base de datos en la estructura de carpetas y ficheros.



2. Comprobar que se tiene el *import* a sqlite correctamente añadido al inicio del programa

```
import sqlite3
```

3. Crear una variable en la clase Producto para acceder a la ruta de la base de datos.

```
class Producto:  
    db = 'database/productos.db'
```

4. Crear un método en la clase Producto que se conecte a la base de datos, ejecute una consulta y cierre la conexión.



```
def db_consulta(self, consulta, parametros = ()):
    with sqlite3.connect(self.db) as con: # Iniciamos una conexion con la base de
datos (alias con)
        cursor = con.cursor() # Generamos un cursor de la conexion para poder
operar en la base de datos
        resultado = cursor.execute(consulta, parametros) # Preparar la consulta
SQL (con parametros si los hay)
        con.commit() # Ejecutar la consulta SQL preparada anteriormente
    return resultado # Retornar el resultado de la consulta SQL
```



15. Implementar el método de get_productos

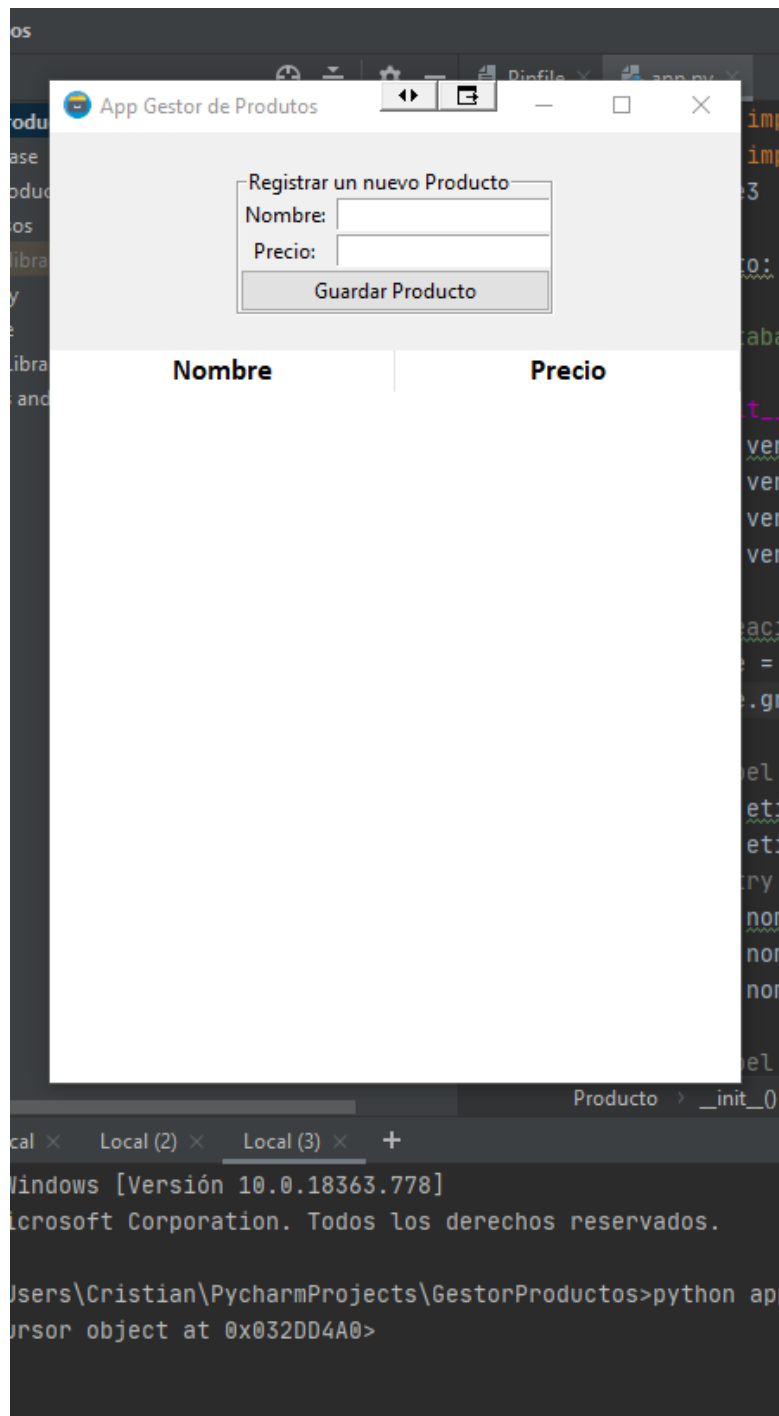
1. Crear un método en la clase Producto que devuelva un listado de todos los productos de la base de datos.

```
def get_productos(self):  
    query = 'SELECT * FROM producto ORDER BY nombre DESC'  
    registros = self.db_consulta(query) # Se hace la llamada al metodo  
db_consultas  
    print(registros) # Se muestran los resultados
```

2. Añadir en el final del constructor de la clase Producto la llamada al método get_productos() el cual nos devolverá el listado de productos al inicio de la aplicación.

```
# Llamada al metodo get_productos() para obtener el listado de productos al  
inicio de la app  
self.get_productos()
```

3. Probar lo implementado. Ejecutar la aplicación.



Comprobar que la información devuelta es el cursor de la base de datos, no los datos en sí.

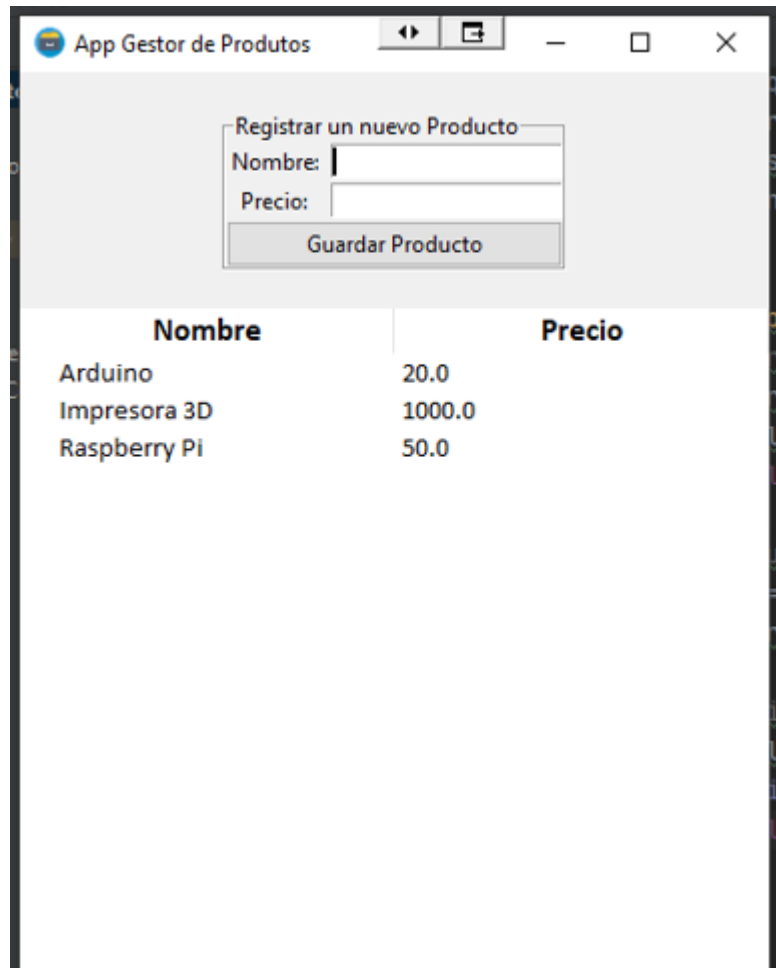
4. Se va a modificar `get_productos()` para poder devolver los datos y no el cursor. Para ello se va a dividir el método en 3 partes:
 - a. Limpiar los datos que pueda tener la tabla.



- b. Realizar la consulta SQL.
- c. Insertar el resultado de la consulta en la tabla.
 - i. Además, para poder verificar los resultados, se incluye un print que muestra los productos por consola.

```
def get_productos(self):  
    # Lo primero, al iniciar la app, vamos a limpiar la tabla por si hubiera  
    # datos residuales o antiguos  
    registros_tabla = self.tabla.get_children() # Obtener todos los datos de la  
    # tabla  
    for fila in registros_tabla:  
        self.tabla.delete(fila)  
  
    # Consulta SQL  
    query = 'SELECT * FROM producto ORDER BY nombre DESC'  
    registros_db = self.db_consulta(query) # Se hace la llamada al metodo  
    # db_consultas  
  
    # Escribir los datos en pantalla  
    for fila in registros_db:  
        print(fila) # print para verificar por consola los datos  
        self.tabla.insert('', 0, text = fila[1], values = fila[2])
```

- 5. Probar lo implementado. Ejecutar la aplicación.



```
Terminal: Local x Local (2) x Local (3) x +
Microsoft Windows [Versión 10.0.18363.778]
(c) 2019 Microsoft Corporation. Todos los derechos reservados.

(venv) C:\Users\Cristian\PycharmProjects\GestorProductos>python app.py
<sqlite3.Cursor object at 0x032DD4A0>

(venv) C:\Users\Cristian\PycharmProjects\GestorProductos>python app.py
(3, 'Raspberry Pi', 50.0)
(1, 'Impresora 3D', 1000.0)
(2, 'Arduino', 20.0)
█
```

16. Implementar el método de add_producto() y sus validaciones

1. Crear un método en la clase Producto que cree un nuevo producto y lo inserte en la base de datos. Este método se llamará add_producto(), pero requerirá de otros métodos secundarios para validar los datos que el usuario ha introducido a través del formulario. Estos métodos se llamarán validacion_nombre() y validacion_precio().

```
def validacion_nombre(self):
    nombre_introducido_por_usuario = self.nombre.get()
    return len(nombre_introducido_por_usuario) != 0

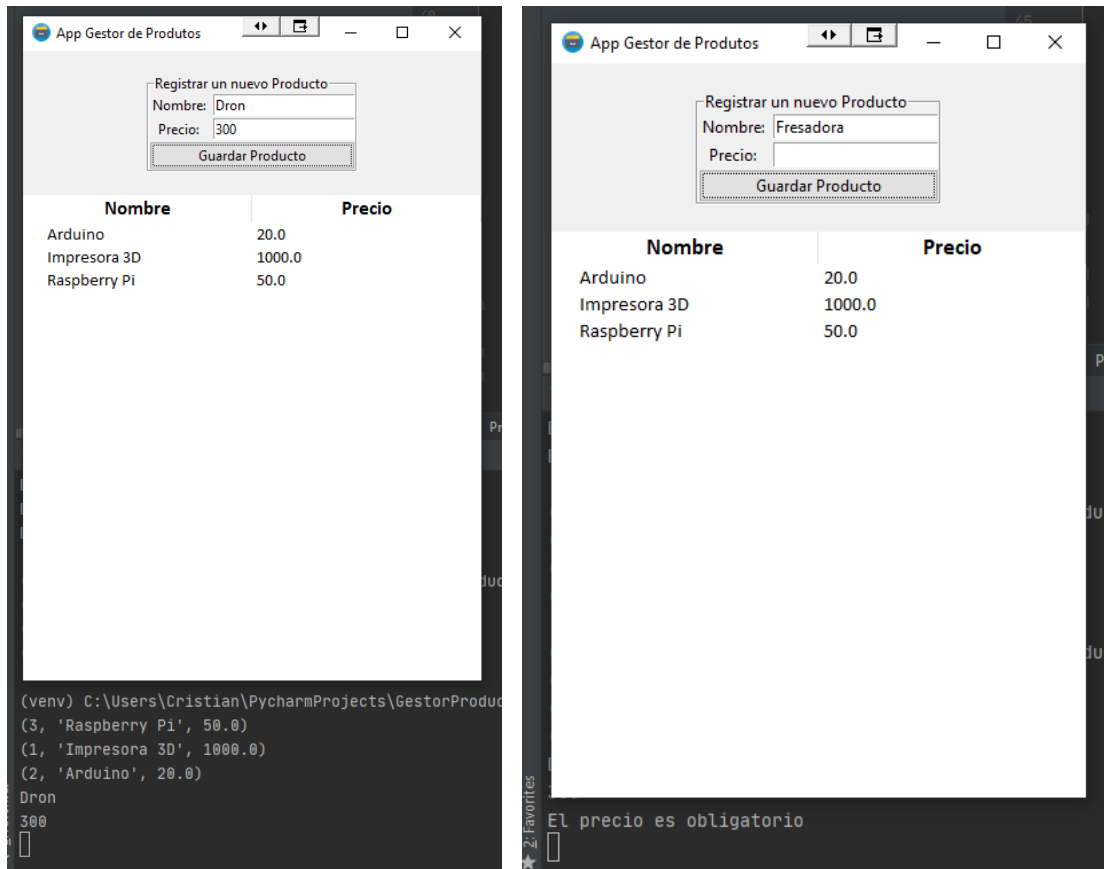
def validacion_precio(self):
    precio_introducido_por_usuario = self.precio.get()
    return len(precio_introducido_por_usuario) != 0

def add_producto(self):
    if self.validacion_nombre() and self.validacion_precio():
        print(self.nombre.get())
        print(self.precio.get())
    elif self.validacion_nombre() and self.validacion_precio() == False:
        print("El precio es obligatorio")
    elif self.validacion_nombre() == False and self.validacion_precio():
        print("El nombre es obligatorio")
    else:
        print("El nombre y el precio son obligatorios")
```

2. Antes de probar el código anteriormente implementado, se debe indicar al botón de Guardar producto debe invocar al método add_producto() al ser clicado. Para ello se vuelve a la definición del botón y se añade el parámetro *command*.

```
# Boton Añadir Producto
self.boton_aniadir = ttk.Button(frame, text = "Guardar Producto", command =
self.add_producto)
```

3. Probar lo implementado. Ejecutar la aplicación



4. Ahora se implementará que esta información se almacene en la base de datos, y que no se muestre por consola simplemente.

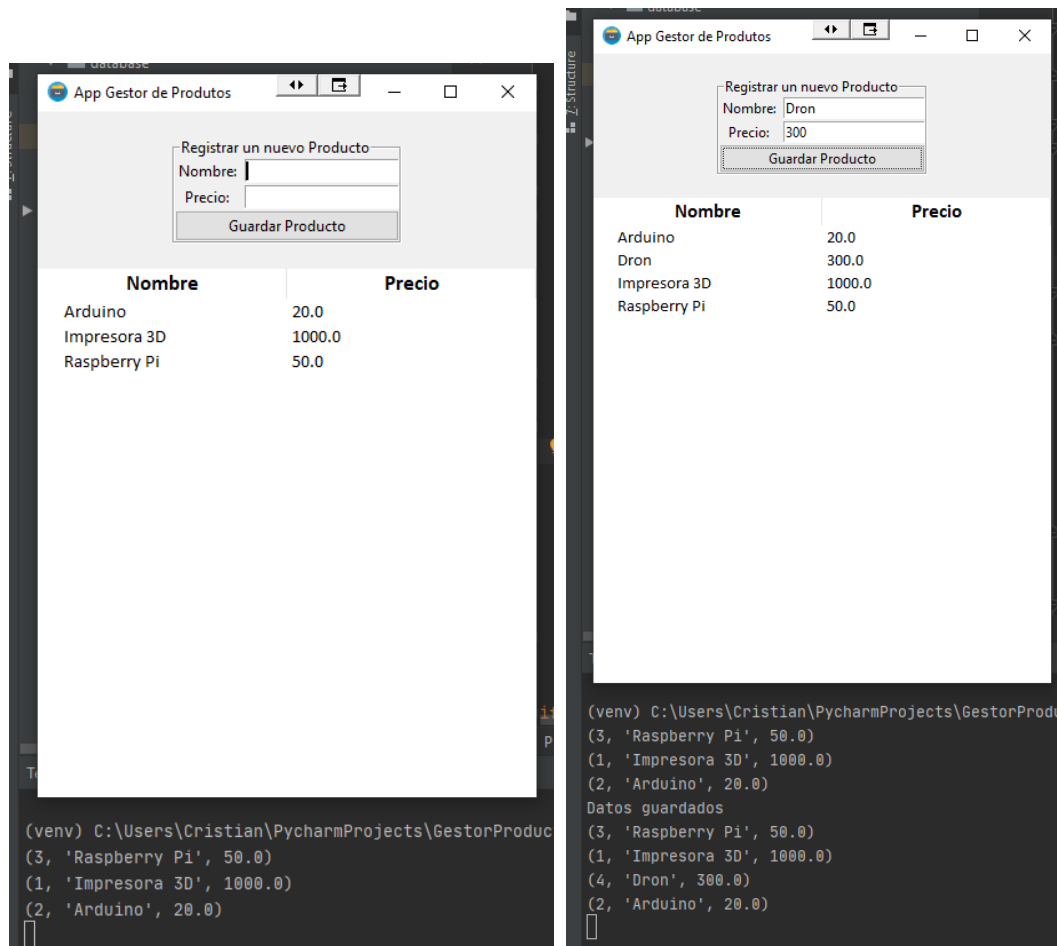
```
def add_producto(self):
    if self.validacion_nombre() and self.validacion_precio():
        query = 'INSERT INTO producto VALUES(NULL, ?, ?)' # Consulta SQL (sin los
datos)
        parametros = (self.nombre.get(), self.precio.get()) # Parametros de la
consulta SQL
        self.db_consulta(query, parametros)
        print("Datos guardados")

        # Para debug
        #print(self.nombre.get())
        #print(self.precio.get())
    elif self.validacion_nombre() and self.validacion_precio() == False:
        print("El precio es obligatorio")
    elif self.validacion_nombre() == False and self.validacion_precio():
        print("El nombre es obligatorio")
    else:
        print("El nombre y el precio son obligatorios")
```



```
self.get_productos() # Cuando se finalice la insercion de datos volvemos a
invocar a este metodo para actualizar el contenido y ver los cambios
```

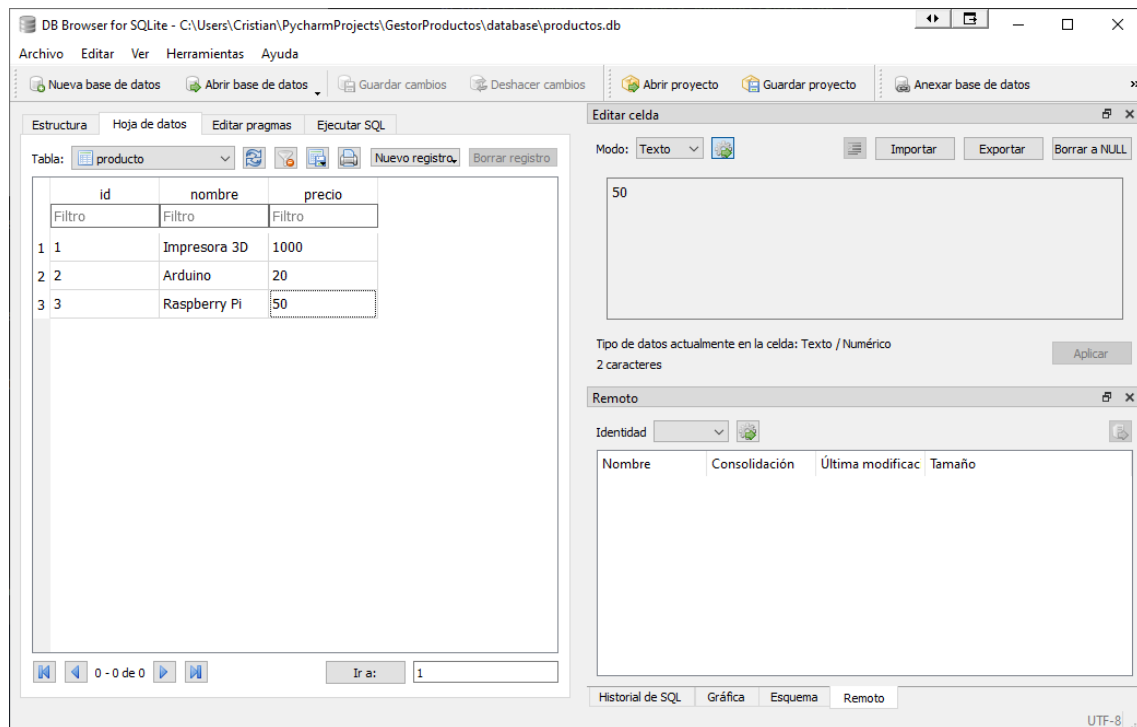
5. Probar lo implementado. Ejecutar la aplicación



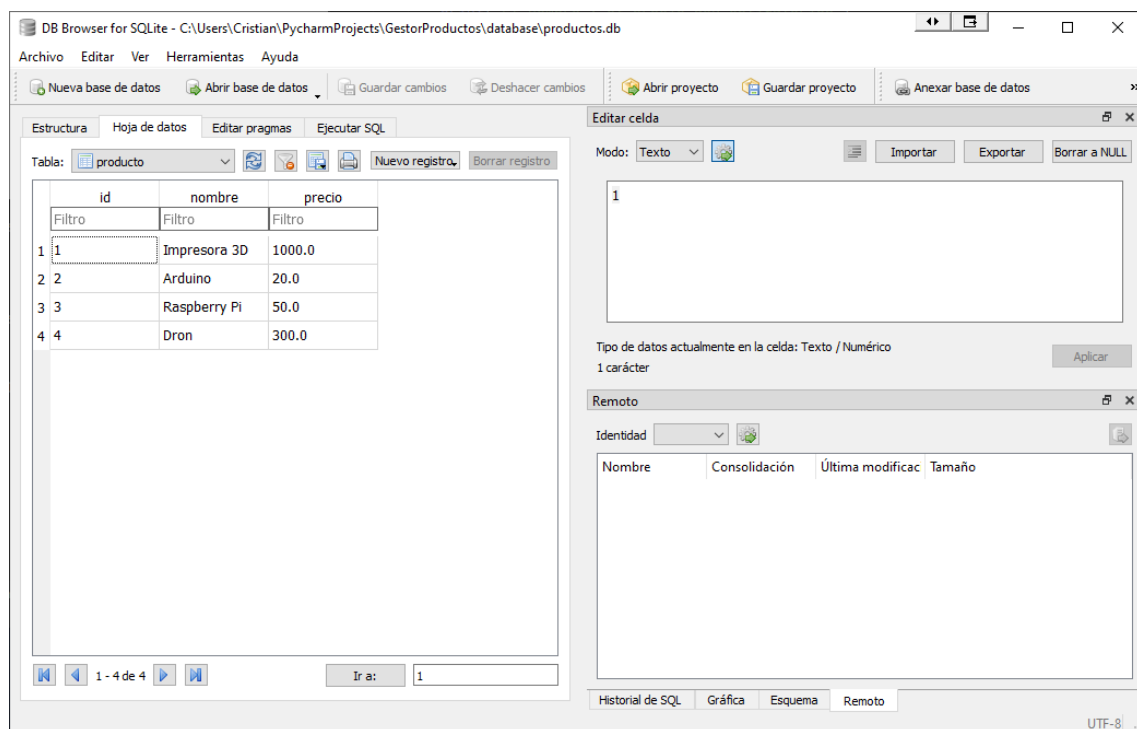
Como se puede apreciar, tras insertar Dron (300) y pulsar en Guardar Producto, se ha añadido a la lista y se puede comprobar visualmente ya que se la aplicación se actualiza tras la inserción llamando al método `get_productos()`. Pero habría que comprobar si en la base de datos todo es correcto.

6. Comprobar en la base de datos a través de DB Browser

Antes:



Después de pulsar el botón de refrescar





17. Mejorando add_producto()

En este punto ya se tiene la funcionalidad de añadir producto perfectamente implementada, pero hay algunos detalles que se pueden mejorar:

- Hacer que cuando se pulse en el botón de Guardar producto.
 - El formulario se limpie automáticamente, para dejarlo listo para otra inserción.
 - Que aparezca un mensaje de confirmación para el usuario.
1. Se vuelve al constructor de la clase Producto, y se va a ubicar un mensaje informativo para el usuario (Label) entre el botón de Guardar Producto y la tabla.

```
# Mensaje informativo para el usuario
self.mensaje = Label(text = '', fg = 'red')
self.mensaje.grid(row = 3, column = 0, colspan = 2, sticky = W + E)
```

Este mensaje inicialmente está vacío, por lo que no se verá nada. Se deberá rellenar con un texto para mostrarse cuando el usuario pulse en Guardar Producto.

2. Se va al método add_producto() y se sustituye el `print("Datos guardados")` por la siguiente instrucción, la cual rellenará el *Label* con un texto para mostrarse.

```
self.mensaje['text'] = 'Producto {} añadido con éxito'.format(self.nombre.get())
# Label ubicado entre el boton y la tabla
```

3. Para hacer que cuando se pulse en Guardar Producto también limpie el formulario, se insertará justo debajo de la línea anterior lo siguiente:

```
self.nombre.delete(0, END) # Borrar el campo nombre del formulario
self.precio.delete(0, END) # Borrar el campo precio del formulario
```

4. Y se modificarán también los mensajes de validación (el precio es obligatorio, el nombre es obligatorio, etc.)

```
self.mensaje['text'] = 'El precio es obligatorio'
...
self.mensaje['text'] = 'El nombre es obligatorio'
...
self.mensaje['text'] = 'El nombre y el precio son obligatorios'
```

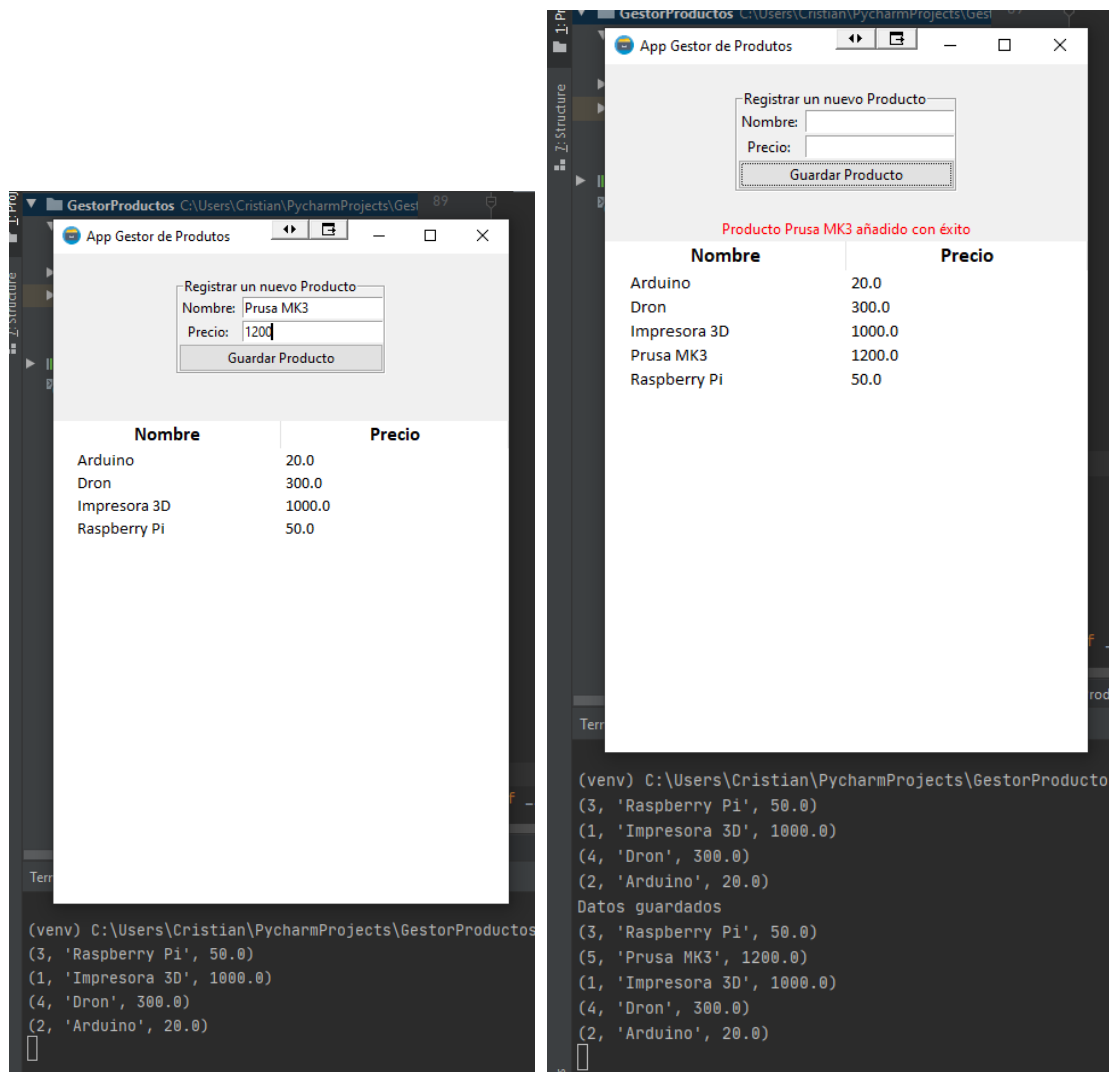
5. El método `add_producto()` quedaría así:

```
def add_producto(self):
    if self.validacion_nombre() and self.validacion_precio():
        query = 'INSERT INTO producto VALUES(NULL, ?, ?)' # Consulta SQL (sin los
datos)
        parametros = (self.nombre.get(), self.precio.get()) # Parametros de la
consulta SQL
        self.db_consulta(query, parametros)
        print("Datos guardados")
        self.mensaje['text'] = 'Producto {} añadido con
éxito'.format(self.nombre.get()) # Label ubicado entre el boton y la tabla
        self.nombre.delete(0, END) # Borrar el campo nombre del formulario
        self.precio.delete(0, END) # Borrar el campo precio del formulario

        # Para debug
        #print(self.nombre.get())
        #print(self.precio.get())
    elif self.validacion_nombre() and self.validacion_precio() == False:
        print("El precio es obligatorio")
        self.mensaje['text'] = 'El precio es obligatorio'
    elif self.validacion_nombre() == False and self.validacion_precio():
        print("El nombre es obligatorio")
        self.mensaje['text'] = 'El nombre es obligatorio'
    else:
        print("El nombre y el precio son obligatorios")
        self.mensaje['text'] = 'El nombre y el precio son obligatorios'

    self.get_productos() # Cuando se finalice la insercion de datos volvemos a
invocar a este metodo para actualizar el contenido y ver los cambios
```

6. Probar lo implementado. Ejecutar la aplicación:



Y probando los mensajes de validación:

App Gestor de Productos

Registrar un nuevo Producto

Nombre:

Precio:

Guardar Producto

El precio es obligatorio

Nombre	Precio
Arduino	20.0
Dron	300.0
Impresora 3D	1000.0
Prusa MK3	1200.0
Raspberry Pi	50.0

App Gestor de Productos

Registrar un nuevo Producto

Nombre:

Precio:

Guardar Producto

El nombre es obligatorio

Nombre	Precio
Arduino	20.0
Dron	300.0
Impresora 3D	1000.0
Prusa MK3	1200.0
Raspberry Pi	50.0

App Gestor de Productos

Registrar un nuevo Producto

Nombre:

Precio:

Guardar Producto

El nombre y el precio son obligatorios

Nombre	Precio
Arduino	20.0
Dron	300.0
Impresora 3D	1000.0
Prusa MK3	1200.0
Raspberry Pi	50.0



18. Añadir los dos botones que falta: Eliminar y Editar

1. Se irá al constructor de la clase Producto, y debajo de la definición de la tabla, se añadirán los dos botones.

```
# Botones de Eliminar y Editar
boton_eliminar = ttk.Button(text = 'ELIMINAR')
boton_eliminar.grid(row = 5, column = 0, sticky = W + E)
boton_editar = ttk.Button(text='EDITAR')
boton_editar.grid(row = 5, column = 1, sticky = W + E)
```

2. Probar lo implementado. Ejecutar la aplicación.

Nombre	Precio
Arduino	20.0
Dron	300.0
Impresora 3D	1000.0
Prusa MK3	1200.0
Raspberry Pi	50.0



19. Implementar la funcionalidad de Eliminar

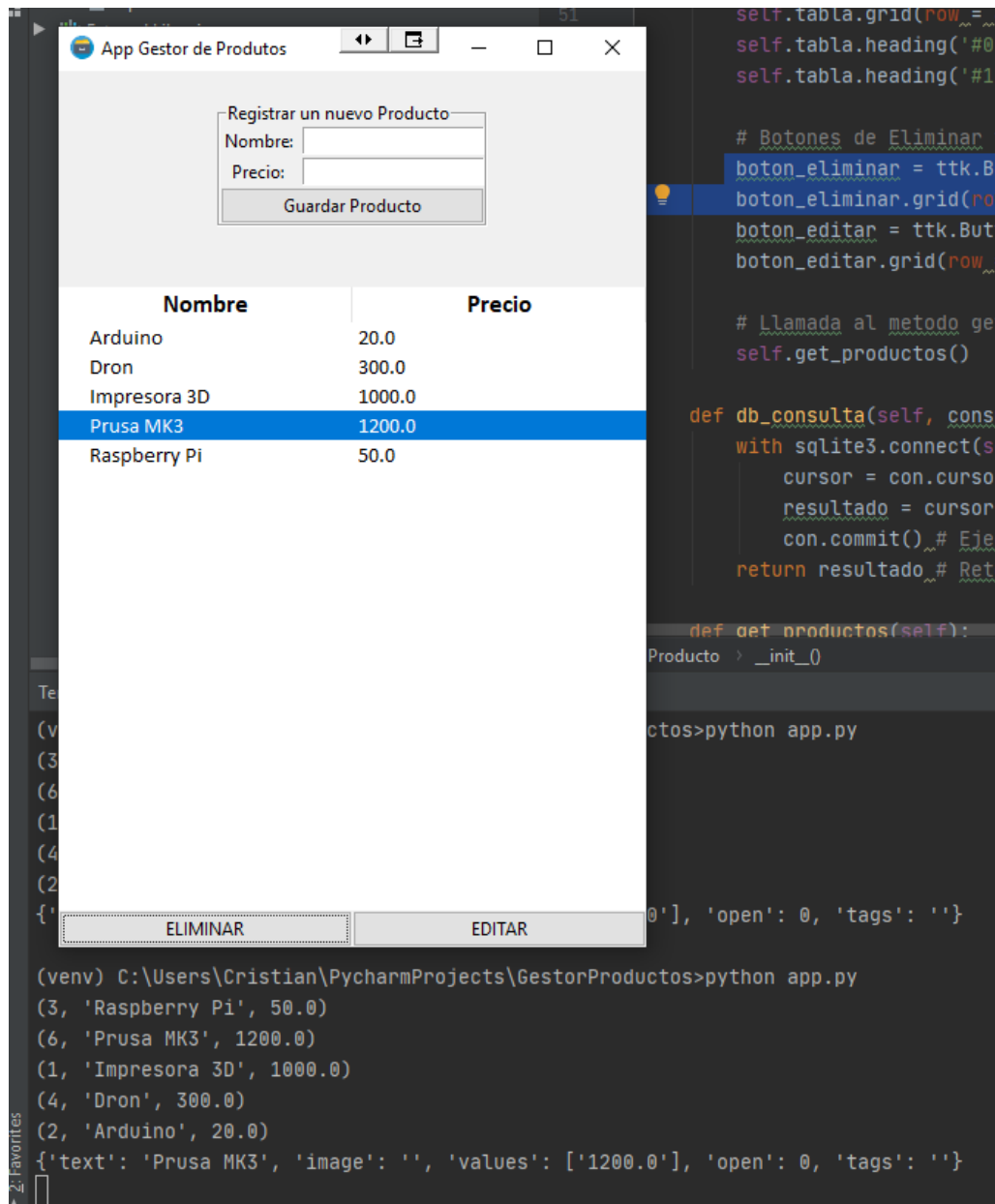
1. Se creara un método llamado `del_producto()`, en la clase `Productos`. De momento se realiza lo siguiente:
 - El usuario selecciona un producto de la tabla.
 - El usuario pulsa el botón `ELIMINAR`.
 - Se muestra por consola la información del producto seleccionado.

```
def del_producto(self):  
    print(self.tabla.item(self.tabla.selection()))
```

2. Se modifica el botón `Eliminar`, añadiendo el atributo `command`.

```
boton_eliminar = ttk.Button(text = 'ELIMINAR', command = self.del_producto)  
boton_eliminar.grid(row = 5, column = 0, sticky = W + E)
```

3. Probar lo implementado. Ejecutar la aplicación. Fijarse en los datos que aparecen en consola tras seleccionar un producto y pulsar `Eliminar`

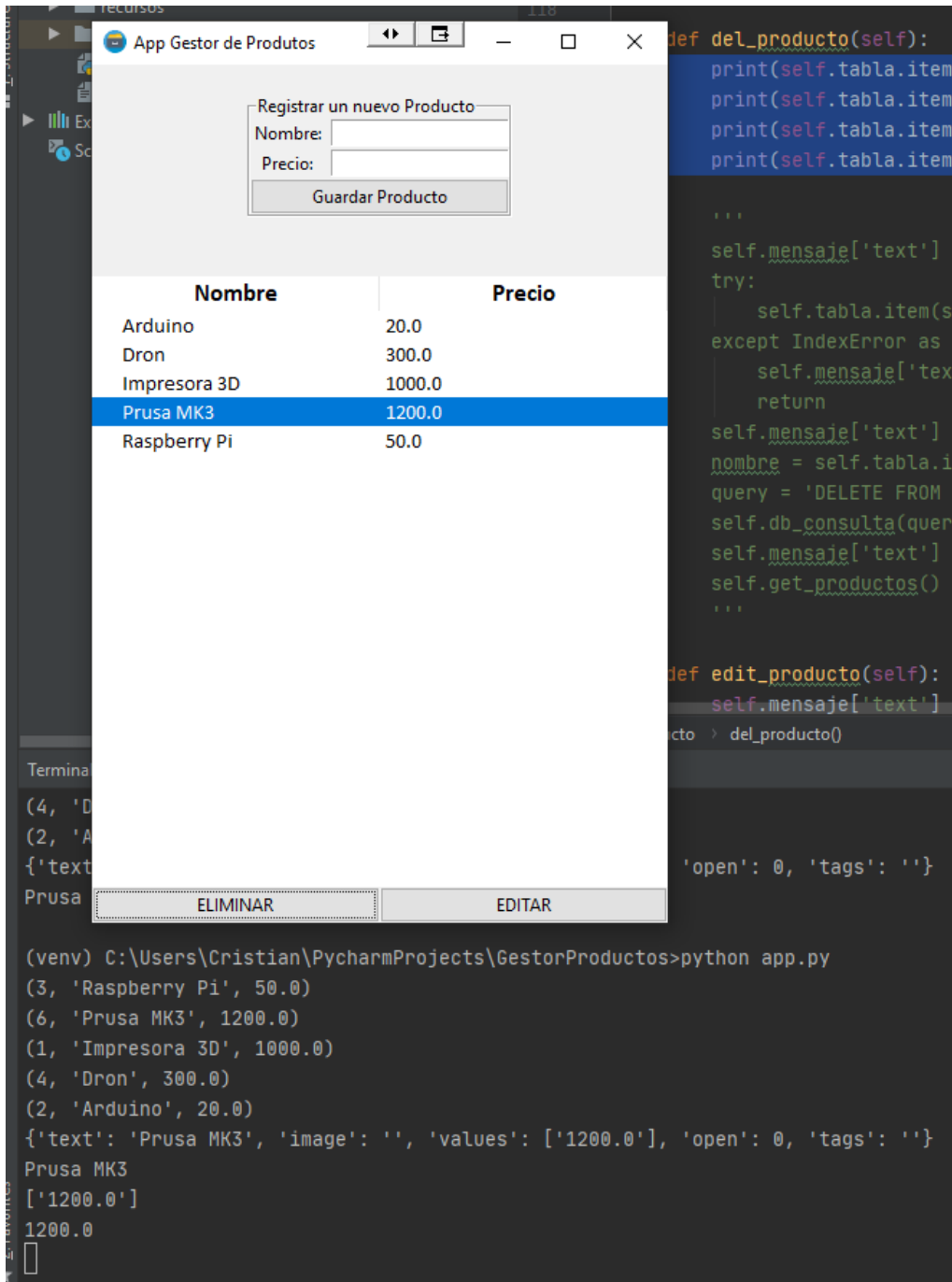


4. En este punto, en el que se está extrayendo toda la información del producto, se va a profundizar un poco para ver cómo obtener el valor del nombre y del precio (que se encuentra dentro de una lista).



```
# Debug
print(self.tabla.item(self.tabla.selection()))
print(self.tabla.item(self.tabla.selection())['text'])
print(self.tabla.item(self.tabla.selection())['values'])
print(self.tabla.item(self.tabla.selection())['values'][0])
```

5. Probar lo implementado. Ejecutar la aplicación. Fijarse en los datos que aparecen en consola tras seleccionar un producto y pulsar Eliminar



Nombre	Precio
Arduino	20.0
Dron	300.0
Impresora 3D	1000.0
Prusa MK3	1200.0
Raspberry Pi	50.0

```
(venv) C:\Users\Cristian\PycharmProjects\GestorProductos>python app.py
(3, 'Raspberry Pi', 50.0)
(6, 'Prusa MK3', 1200.0)
(1, 'Impresora 3D', 1000.0)
(4, 'Dron', 300.0)
(2, 'Arduino', 20.0)
{'text': 'Prusa MK3', 'image': '', 'values': ['1200.0'], 'open': 0, 'tags': ''}
Prusa MK3
['1200.0']
1200.0
█
```

- Ahora que el botón funciona, y se tiene acceso a la información, se pasa a realizar la implementación final. Se comentarán las líneas anteriores para conservarlas para *debug*, y se añadirá lo siguiente:



```
def del_producto(self):
    # Debug
    #print(self.tabla.item(self.tabla.selection()))
    #print(self.tabla.item(self.tabla.selection())['text'])
    #print(self.tabla.item(self.tabla.selection())['values'])
    #print(self.tabla.item(self.tabla.selection())['values'][0])

    self.mensaje['text'] = '' # Mensaje inicialmente vacío
    # Comprobación de que se seleccione un producto para poder eliminarlo
    try:
        self.tabla.item(self.tabla.selection())['text'][0]
    except IndexError as e:
        self.mensaje['text'] = 'Por favor, seleccione un producto'
        return

    self.mensaje['text'] = ''
    nombre = self.tabla.item(self.tabla.selection())['text']
    query = 'DELETE FROM producto WHERE nombre = ?' # Consulta SQL
    self.db_consulta(query, (nombre,)) # Ejecutar la consulta
    self.mensaje['text'] = 'Producto {} eliminado con éxito'.format(nombre)
    self.get_productos() # Actualizar la tabla de productos
```

7. Probar lo implementado. Ejecutar la aplicación.

App Gestor de Productos

Registrar un nuevo Producto

Nombre:

Precio:

Guardar Producto

Nombre	Precio
Arduino	20.0
Dron	300.0
Impresora 3D	1000.0
Prusa MK3	1200.0
Raspberry Pi	50.0

ELIMINAR EDITAR

```

# print(self.t
# print(self.t
# print(self.t

self.mensaje[
# Comprobacio
try:
    self.tabl
except IndexE
    self.mens
    return

self.mensaje[
nombre = self
query = 'DELE
self.db_consu
self.mensaje[
self.get_pro

def edit_producto
Producto -> del_producto()

uctos>python app.py

(venv) C:\Users\Cristian\PycharmProjects\GestorProductos>python app.py
(3, 'Raspberry Pi', 50.0)
(6, 'Prusa MK3', 1200.0)
(1, 'Impresora 3D', 1000.0)
(4, 'Dron', 300.0)
(2, 'Arduino', 20.0)
  
```

App Gestor de Productos

Registrar un nuevo Producto

Nombre:

Precio:

Guardar Producto

Producto Prusa MK3 eliminado con éxito

Nombre	Precio
Arduino	20.0
Dron	300.0
Impresora 3D	1000.0
Raspberry Pi	50.0

ELIMINAR EDITAR

Base de datos antes:

Tabla: producto

	id	nombre	precio
	Filtro	Filtro	Filtro
1	1	Impresora 3D	1000.0
2	2	Arduino	20.0
3	3	Raspberry Pi	50.0
4	4	Dron	300.0
5	6	Prusa MK3	1200.0

Base de datos después:

Tabla: producto

	id	nombre	precio
	Filtro	Filtro	Filtro
1	1	Impresora 3D	1000.0
2	2	Arduino	20.0
3	3	Raspberry Pi	50.0
4	4	Dron	300.0

20. Implementar la funcionalidad de Editar

1. Se creará un método llamado `edit_producto()` dentro de la clase `Producto`. Será muy similar a eliminar producto, con la diferencia de que crearemos una segunda ventana para poder modificar los datos de un producto:

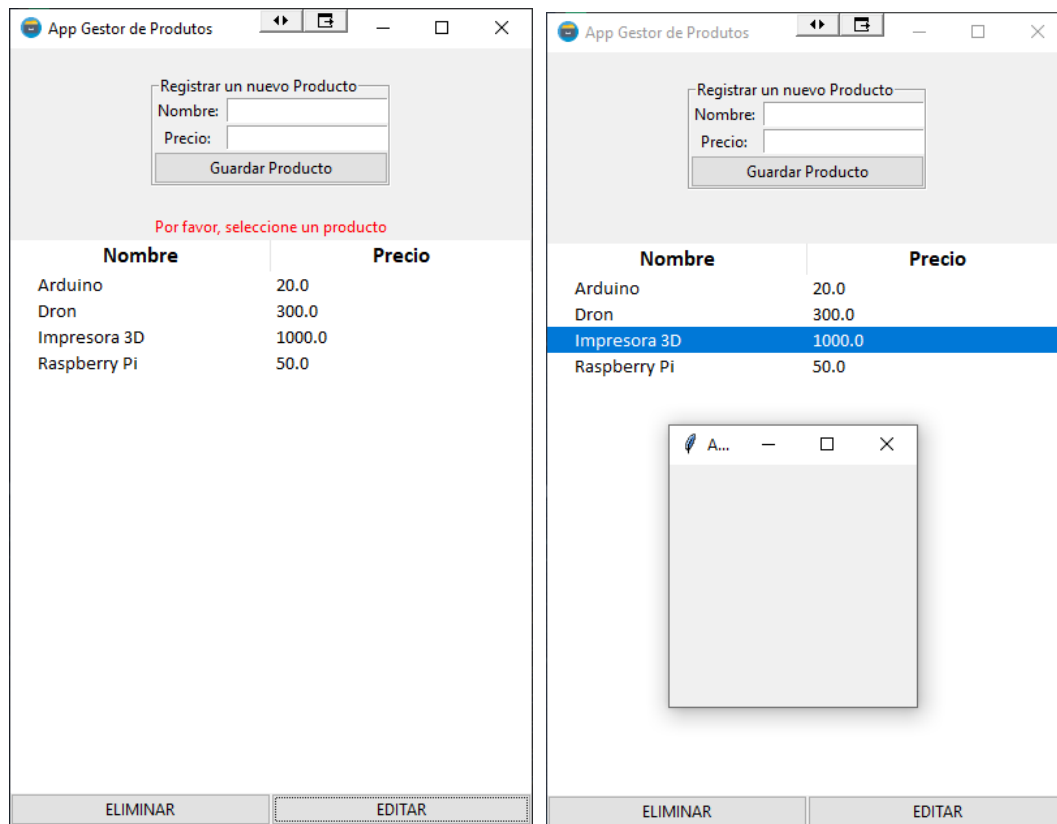
```
def edit_producto(self):
    self.mensaje['text'] = '' # Mensaje inicialmente vacío
    try:
        self.tabla.item(self.tabla.selection())['text'][0]
    except IndexError as e:
        self.mensaje['text'] = 'Por favor, seleccione un producto'
    return
    nombre = self.tabla.item(self.tabla.selection())['text']
    old_precio = self.tabla.item(self.tabla.selection())['values'][0] # El precio
    se encuentra dentro de una lista

    self.ventana_editar = Toplevel() # Crear una ventana por delante de la
    principal
    self.ventana_editar.title = "Editar Producto" # Título de la ventana
    self.ventana_editar.resizable(1, 1) # Activar la redimension de la ventana.
    Para desactivarla: (0,0)
    self.ventana_editar.wm_iconbitmap('recursos/icon.ico') # Icono de la ventana
```

2. Se modifica el botón Editar, añadiendo el atributo `command`

```
boton_editar = ttk.Button(text='EDITAR', command = self.edit_producto)
```

3. Probar lo implementado. Ejecutar la aplicación.



4. Justo a continuación de la creación de la segunda ventana, se añadirá el siguiente código que cubre los siguientes aspectos:

- Un título para esta nueva ventana.
- Un *frame* que englobe:
 - El nombre antiguo (sin que se pueda modificar).
 - El precio antiguo (sin que se pueda modificar).
 - El nombre nuevo.
 - El precio nuevo.
 - Un botón para actualizar el producto.

El comportamiento ideal sería:

- Si el usuario modifica el nombre y el precio:
 - Se modifica nombre y precio (tanto en la tabla como en la base de datos).
- Si el usuario modifica solo el nombre:
 - Se modifica sólo el nombre, conservando el precio antiguo.
- Si el usuario modifica solo el precio:

- Se modifica sólo el precio, conservando el nombre antiguo
- Si el usuario no modifica nada:
 - No se modifica nada, se conserva nombre y precio antiguo

En todos los casos, se deberá notificar de los cambios al usuario con un texto en pantalla.

```
# Ventana nueva (editar producto)
self.ventana_editar = Toplevel() # Crear una ventana por delante de la principal
self.ventana_editar.title = "Editar Producto" # Título de la ventana
self.ventana_editar.resizable(1, 1) # Activar la redimension de la ventana. Para
desactivarla: (0,0)
self.ventana_editar.wm_iconbitmap('recursos/icon.ico') # Icono de la ventana

titulo = Label(self.ventana_editar, text='Edición de Productos', font=('Calibri',
50, 'bold'))
titulo.grid(column=0, row=0)

# Creacion del contenedor Frame de la ventana de Editar Producto
frame_ep = LabelFrame(self.ventana_editar, text="Editar el siguiente Producto") #
frame_ep: Frame Editar Producto
frame_ep.grid(row=1, column=0, columnspan=20, pady=20)

# Label Nombre antiguo
self.etiqueta_nombre_antiguo = Label(frame_ep, text = "Nombre antiguo: ") #
Etiqueta de texto ubicada en el frame
self.etiqueta_nombre_antiguo.grid(row=2, column=0) # Posicionamiento a traves
de grid
# Entry Nombre antiguo (texto que no se podra modificar)
self.input_nombre_antiguo = Entry(frame_ep,
textvariable=StringVar(self.ventana_editar, value=nombre), state='readonly')
self.input_nombre_antiguo.grid(row=2, column=1)

# Label Nombre nuevo
self.etiqueta_nombre_nuevo = Label(frame_ep, text="Nombre nuevo: ")
self.etiqueta_nombre_nuevo.grid(row=3, column=0)
# Entry Nombre nuevo (texto que si se podra modificar)
self.input_nombre_nuevo = Entry(frame_ep)
self.input_nombre_nuevo.grid(row=3, column=1)
self.input_nombre_nuevo.focus() # Para que el foco del raton vaya a este Entry al
inicio

# Label Precio antiguo
self.etiquetaPrecio_antiguo = Label(frame_ep, text="Precio antiguo: ") #
Etiqueta de texto ubicada en el frame
self.etiquetaPrecio_antiguo.grid(row=4, column=0) # Posicionamiento a traves
de grid
# Entry Precio antiguo (texto que no se podra modificar)
self.input_precio_antiguo = Entry(frame_ep,
textvariable=StringVar(self.ventana_editar, value=old_precio), state='readonly')
self.input_precio_antiguo.grid(row=4, column=1)
```

```
# Label Precio nuevo
self.etiqueta_precio_nuevo = Label(frame_ep, text="Precio nuevo: ")
self.etiqueta_precio_nuevo.grid(row=5, column=0)
# Entry Precio nuevo (texto que si se podra modificar)
self.input_precio_nuevo = Entry(frame_ep)
self.input_precio_nuevo.grid(row=5, column=1)

# Boton Actualizar Producto
self.boton_actualizar = ttk.Button(frame_ep, text="Actualizar Producto",
                                   command=lambda:
self.actualizar_productos(self.input_nombre_nuevo.get(),
self.input_nombre_antiguo.get(),
self.input_precio_nuevo.get(),
self.input_precio_antiguo.get()))

self.boton_actualizar.grid(row=6, columnspan=2, sticky=W + E)
```

5. Y finalmente, se implementa el método `actualizar_productos()` el cual es llamado desde `edit_producto()`, y que tiene como objetivo:

- Ejecutar la consulta SQL.
- Modificar el texto que aparece en pantalla para retroalimentar al usuario.
- Cerrar la ventana de editar productos.
- Actualizar la tabla de productos de la aplicación (consultando de nuevo a la base de datos).

```
def actualizar_productos(self, nuevo_nombre, antiguo_nombre, nuevo_precio,
antiguo_precio):
    producto_modificado = False
    query = 'UPDATE producto SET nombre = ?, precio = ? WHERE nombre = ? AND
precio = ?'
    if nuevo_nombre != '' and nuevo_precio != '':
        # Si el usuario escribe nuevo nombre y nuevo precio, se cambian ambos
        parametros = (nuevo_nombre, nuevo_precio, antiguo_nombre, antiguo_precio)
        producto_modificado = True
    elif nuevo_nombre != '' and nuevo_precio == '':
        # Si el usuario deja vacio el nuevo precio, se mantiene el precio anterior
        parametros = (nuevo_nombre, antiguo_precio, antiguo_nombre,
antiguo_precio)
        producto_modificado = True
    elif nuevo_nombre == '' and nuevo_precio != '':
        # Si el usuario deja vacio el nuevo nombre, se mantiene el nombre
anterior
        parametros = (antiguo_nombre, nuevo_precio, antiguo_nombre,
antiguo_precio)
        producto_modificado = True

    if(producto_modificado):
```



```
self.db_consulta(query, parametros) # Ejecutar la consulta
self.ventana_editar.destroy() # Cerrar la ventana de edicion de productos
self.mensaje['text'] = 'El producto {} ha sido actualizado con
éxito'.format(antiguo_nombre) # Mostrar mensaje para el usuario
self.get_productos() # Actualizar la tabla de productos
else:
self.ventana_editar.destroy() # Cerrar la ventana de edicion de
productos
self.mensaje['text'] = 'El producto {} NO ha sido
actualizado'.format(antiguo_nombre) # Mostrar mensaje para el usuario
```

6. Probar lo implementado. Ejecutar la aplicación.



App Gestor de Productos

Registrar un nuevo Producto

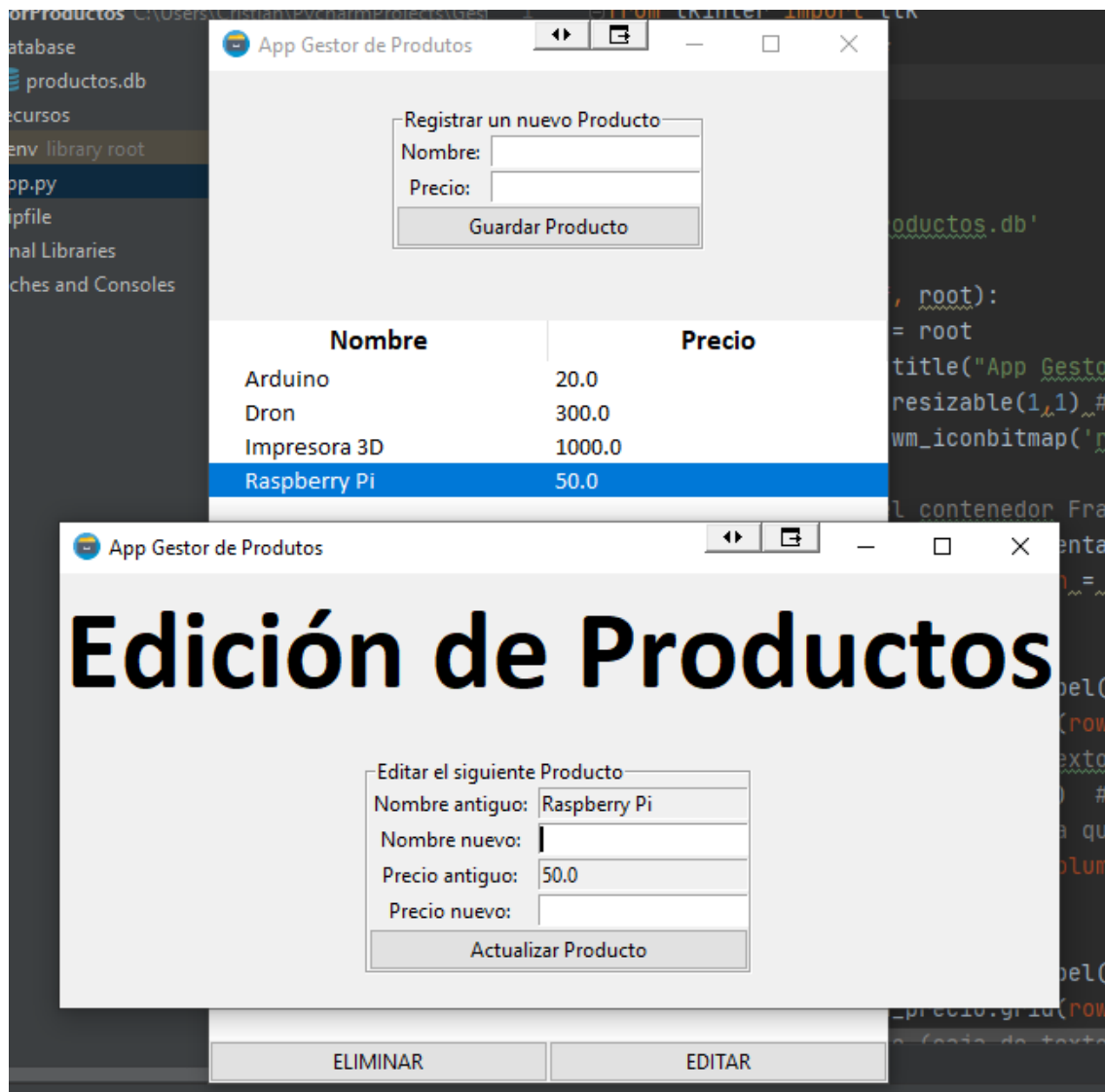
Nombre:

Precio:

Guardar Producto

Nombre	Precio
Arduino	20.0
Dron	300.0
Impresora 3D	1000.0
Raspberry Pi	50.0

ELIMINAR EDITAR



Se modifica el producto y se pulsa en Actualizar Producto.



App Gestor de Productos

Edición de Productos

Editar el siguiente Producto

Nombre antiguo:	Raspberry Pi
Nombre nuevo:	Raspberry Pi 4
Precio antiguo:	50.0
Precio nuevo:	60

Actualizar Producto

Resultado final en la aplicación:



App Gestor de Productos

Registrar un nuevo Producto

Nombre:

Precio:

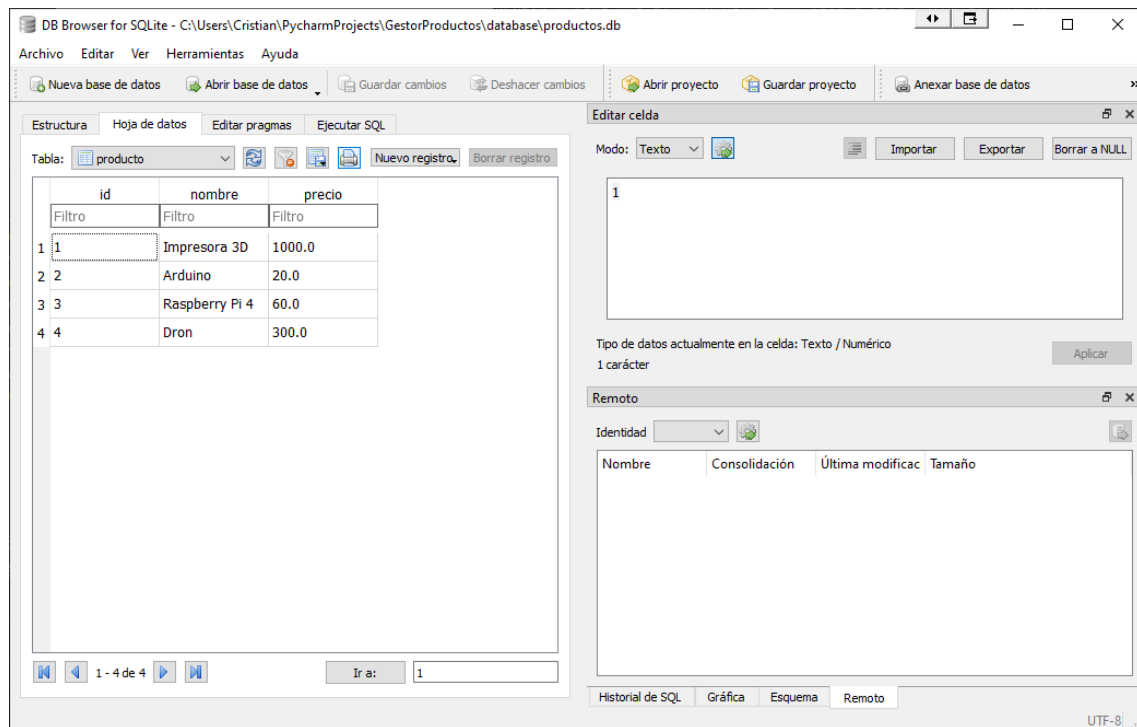
Guardar Producto

El producto Raspberry Pi ha sido actualizado con éxito

Nombre	Precio
Arduino	20.0
Dron	300.0
Impresora 3D	1000.0
Raspberry Pi 4	60.0

ELIMINAR EDITAR

Resultado final en la base de datos (si no se ven los cambios a priori actualizar el contenido):



Posibles evolutivos de esta aplicación:

- Implementar SQLAlchemy sobre SQLite para poder cambiar de base de datos sin necesidad de tocar el código.
- Añadir una sección más a los Productos, que sea la Categoría. Para ello habrá que modificar el formulario de inserción de productos, el listado de productos, el modelo de la base de datos, etc.
- Agregar algún nuevo widget gráfico proporcionado por Tkinter, que no sea una *label*, un botón, un cajón de texto o una tabla.

21. Mejorando el diseño

1. Se aplicará algo de estilo a los diferentes textos que se tienen en la aplicación. Añadir el siguiente estilo:

```
font=('Calibri', 16, 'bold')
```

A los textos simples más importantes:

```
frame = LabelFrame(self.ventana, text = "Registrar un nuevo Producto",  
font=('Calibri', 16, 'bold'))  
  
frame_ep = LabelFrame(self.ventana_editar, text="Editar el siguiente Producto",  
font=('Calibri', 16, 'bold'))
```

Y el siguiente estilo al resto de textos (nombre, precio, etc.)

```
font=('Calibri', 13)  
  
self.etiqueta_nombre = Label(frame, text="Nombre: ", font=('Calibri', 13))  
self.nombre = Entry(frame, font=('Calibri', 13))  
  
self.etiqueta_precio = Label(frame, text="Precio: ", font=('Calibri', 13))  
self.precio = Entry(frame, font=('Calibri', 13))  
  
self.etiqueta_nombre_antiguo = Label(frame_ep, text = "Nombre antiguo: ",  
font=('Calibri', 13))  
  
self.input_nombre_antiguo = Entry(frame_ep,  
textvariable=StringVar(self.ventana_editar, value=nombre), state='readonly',  
font=('Calibri', 13))  
  
self.etiqueta_nombre_nuevo = Label(frame_ep, text="Nombre nuevo: ",  
font=('Calibri', 13))  
  
self.input_nombre_nuevo = Entry(frame_ep, font=('Calibri', 13))  
  
self.etiqueta_precio_antiguo = Label(frame_ep, text="Precio antiguo: ",  
font=('Calibri', 13))
```



```
self.input_precio_antiguo = Entry(frame_ep,  
textvariable=StringVar(self.ventana_editar, value=old_precio),state='readonly',  
font=('Calibri', 13))  
  
self.etiqueta_precio_nuevo = Label(frame_ep, text="Precio nuevo: ",  
font=('Calibri', 13))  
  
self.input_precio_nuevo = Entry(frame_ep, font=('Calibri', 13))
```

2. Modificar los botones de la siguiente manera:

```
# Boton Añadir Producto  
s = ttk.Style()  
s.configure('my.TButton', font=('Calibri', 14, 'bold'))  
self.boton_aniadir = ttk.Button(frame, text="Guardar Producto",  
command=self.add_producto, style='my.TButton')  
self.boton_aniadir.grid(row=3, columnspan=2, sticky=W + E)
```

```
# Botones de Eliminar y Editar  
s = ttk.Style()  
s.configure('my.TButton', font=('Calibri', 14, 'bold'))  
boton_eliminar = ttk.Button(text = 'ELIMINAR', command = self.del_producto,  
style='my.TButton')  
boton_eliminar.grid(row = 5, column = 0, sticky = W + E)  
boton_editar = ttk.Button(text='EDITAR', command = self.edit_producto,  
style='my.TButton')  
boton_editar.grid(row = 5, column = 1, sticky = W + E)
```

```
# Boton Actualizar Producto  
s = ttk.Style()  
s.configure('my.TButton', font=('Calibri', 14, 'bold'))  
self.boton_actualizar = ttk.Button(frame_ep, text="Actualizar Producto",  
style='my.TButton',  
command=lambda:  
self.actualizar_productos(self.input_nombre_nuevo.get(),  
self.input_nombre_antiguo.get(),  
self.input_precio_nuevo.get(),  
self.input_precio_antiguo.get()))  
self.boton_actualizar.grid(row=6, columnspan=2, sticky=W + E)
```



22. Resultado final

App Gestor de Productos

Registrar un nuevo Producto

Nombre:

Precio:

Guardar Producto

Nombre	Precio
Arduino	20.0
Dron	300.0
Impresora 3D	1000.0
Raspberry Pi 4	60.0

ELIMINAR **EDITAR**



App Gestor de Productos

Edición de Productos

Editar el siguiente Producto

Nombre antiguo: Raspberry Pi 4

Nombre nuevo:

Precio antiguo: 60.0

Precio nuevo:

Actualizar Producto

O si se quitase el título de Edición de Productos, el resultado final sería:



App Gestor de Productos

Registrar un nuevo Producto

Nombre:

Precio:

Guardar Producto

Nombre	Precio
Arduino	20.0
Dron	300.0
Impresora 3D	1000.0
Raspberry Pi 4	60.0

App Gestor de P

Editar el siguiente Producto

Nombre antiguo: Raspberry Pi 4

Nombre nuevo:

Precio antiguo: 60.0

Precio nuevo:

Actualizar Producto

ELIMINAR

EDITAR



23. Mejoras y entrega de la práctica

Mejoras que se pueden realizar a la práctica:

- Cambiar y mejorar la interfaz gráfica.
- Añadir un campo Categoría para el producto.
- Añadir un campo de Stock para el producto.

Entrega de la práctica (todos estos puntos son obligatorios):

- Realizar un documento de texto con capturas de pantalla donde se vea el funcionamiento completo de la app (y de la base de datos)
- Comprimir en un fichero la carpeta completa del proyecto de Pycharm y el documento de texto y llamarlo: M6_02_nombre_apellido1_apellido2.zip (cambiando nombre y apellidos por los tuyos).



24. Bibliografía

Python 3

<https://www.python.org/>

IDE Pycharm Community

<https://www.jetbrains.com/es-es/pycharm/download/#section=windows>

Módulo Tkinter (web oficial)

<https://docs.python.org/3/library/tkinter.html>

SQLite (documentación oficial)

<https://www.sqlite.org/index.html>

DB Browser for SQLite (interfaz gráfica para SQLite)

<https://sqlitebrowser.org/>

Manual (no oficial) de Tkinter

<https://guia-tkinter.readthedocs.io/es/develop/>

Manual (no oficial) de Tkinter. Sección widgets

<https://guia-tkinter.readthedocs.io/es/develop/chapters/6-widgets/6.1-Intro.html#etiquetas-label>