

# PYTHON

APP DE ESCRITÓRIO COM  
THINKER E BBDD

PRÁTICA 2-M6



## Prática de criação de um desktop app com ligação à base de dados

Nesta prática, vai ser criado um desktop app com conexão a uma base de dados. Esta aplicação tem como objetivo ser um gestor de produtos, ou seja, uma aplicação que permite ao utilizador fazer as seguintes ações:

- Criar um produto (nome e preço)
- Editar um produto
- Eliminar um produto

O **stack tecnológico** que se vai utilizar neste projeto é o seguinte:

- **Python 3.** Como linguagem de programação base.
- **Jetbrains Pycharm Community.** IDE escolhido para o desenvolvimento do projeto.
- **Tkinter.** Módulo integrado em Python que proporciona interfaces gráficas.
- **SQLite.** Base de dados SQL rápida e potente para instalações de tamanho moderado.
- **Virtualenv.** Ambiente virtual de Python onde se irá programar o projeto.

**Pré-requisitos** do projeto:

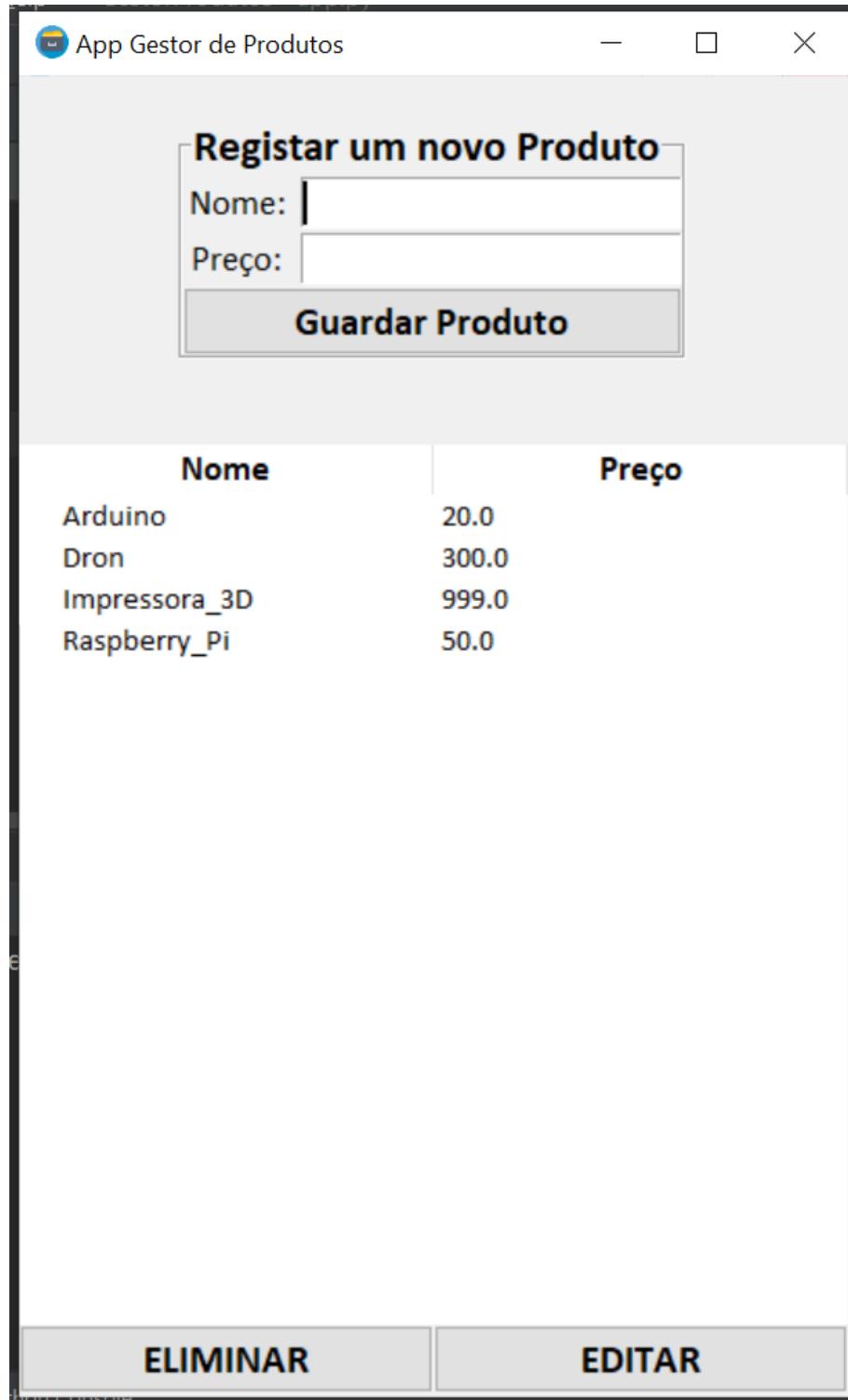
- **Python 3** instalado.
- **Pycharm** instalado.
- **SQLite** instalado.

O objetivo da prática não é criar uma aplicação esteticamente bonita, já que o **Tkinter** é forte na simplicidade ao invés da estética. O objetivo é criar uma aplicação de gestão, simples, útil e funcional.

É impossível abordar uma explicação completa do módulo Tkinter neste projeto, por isso, na bibliografia encontra-se a ligação para a documentação oficial (muito técnica) e um guia (não oficial) que explica a estrutura e componentes gráficos de Tkinter em espanhol.



Para terminar com esta introdução, ver um resumo do resultado deste projeto:





App Gestor de Produtos

# Edição de Produtos

Editar o seguinte Produto

Nome antigo: Dron

Nome novo:

Preço antigo: 300.0

Preço novo:

**Atualizar Produto**

New Database Open Database Write Changes Revert Changes Open Project Save

Database Structure Browse Data Edit Pragmas Execute SQL

Table: produto

	id	nome	preco
Filter	1	Impressora_3D	999.0
2	2	Arduino	20.0
3	3	Raspberry_Pi	50.0
4	8	Dron	300.0



## Conteúdo

1.	Criação do projeto e integração num ambiente virtual	5
2.	Instalação de módulos dentro do ambiente virtual	10
3.	Sair e entrar no ambiente virtual	10
4.	Criação do ficheiro Python principal e primeira janela	11
5.	Começar a implementação usando Classes e Objetos	15
6.	Configuração base da janela principal	16
7.	Estrutura dos widgets (componentes gráficos) na janela	18
8.	Widgets utilizados neste projeto	21
9.	Posicionamento dos elementos em Tkinter	24
10.	Começar com a interface gráfica Adicionar produto	29
11.	Interface gráfica. Tabela de produtos	33
12.	Criação da base de dados	35
13.	Inserir dados de teste na base de dados	40
14.	Implementar a conexão à base de dados desde Python	42
15.	Implementar o método de get_produtos	44
16.	Implementar o método de add_produto() e as suas validações	48
17.	Melhorar add_produto()	52
18.	Adicionar os dois botões que faltam: Eliminar e Editar	56
19.	Implementar a funcionalidade de Eliminar	57
20.	Implementar a funcionalidade de Editar	64
21.	Melhorar o desenho	73
22.	Resultado final	75
23.	Bibliografia	78



## 1. Criação do projeto e integração num ambiente virtual

Num mundo ideal, trabalha-se em todos os projetos com a mesma versão de Python e com os mesmos módulos ou livrarias. Mas a realidade é muito distinta, cada projeto é totalmente diferente e utiliza versões de Python ou versões de módulos ou livrarias diferentes. Por isso, se instalar num sistema a versão de Python 3.6.2 por exemplo e uns módulos ou livrarias determinadas, todos os projetos teriam de utilizar essa versão de Python como essa listagem de módulos e livrarias instaladas. Evidentemente, isto não é funcional nem prático. Por isso, **Python dispõe dos ambientes virtuais**, o que proporciona criar um ambiente totalmente novo e limpo para cada projeto. Podendo desta forma, ter num único sistema, num único equipamento, muitos ambientes virtuais para muitos projetos e onde cada ambiente virtual estará configurado de uma maneira. Exemplo:

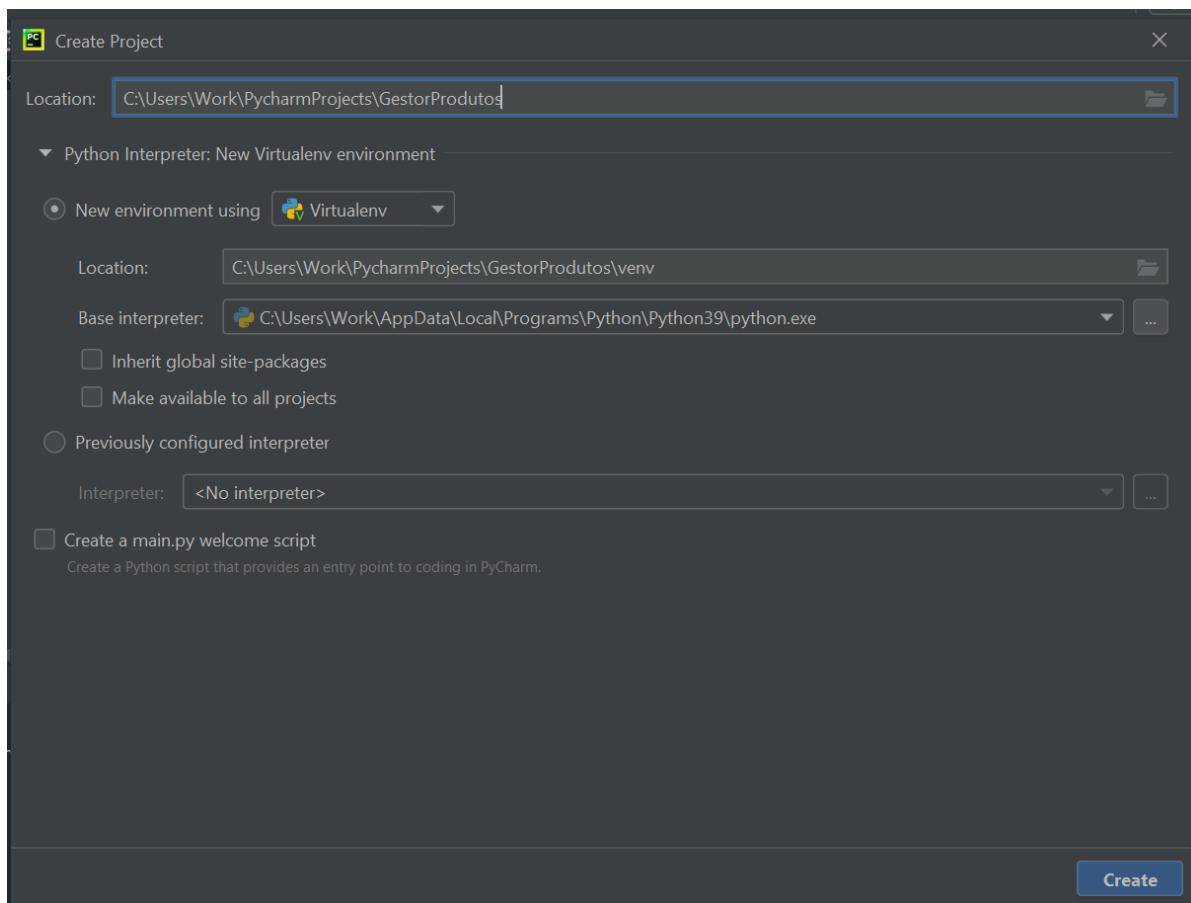
- Ambiente visual 1: Python 3.6.2 com o módulo SQLAlchemy (v2.5) e Pandas (v1.2)
- Ambiente visual 2: Python 3.1 com o módulo SQLAlchemy (v2.0) e Pandas (v1.2)
- Etc.

Esta é a forma na qual se trabalha profissionalmente, utilizando ambientes virtuais para os projetos. Pelo que se vai criar este projeto seguindo esta metodologia.

1. Abrir o IDE de Python com o qual se irá programar. Neste caso, será **Pycharm**
2. Criar um novo projeto
  - File > New Project... >
  - Indicar localização e nome do projeto, neste caso, **GestorProdutos** e na localização por defeito, na pasta de projetos de PyCharm
3. Selecionar “**New environment using > Virtualenv**”



**Virtualenv** é a ferramenta por defeito para criar ambientes virtuais.



Neste ponto está criado o projeto, embora vazio de momento.

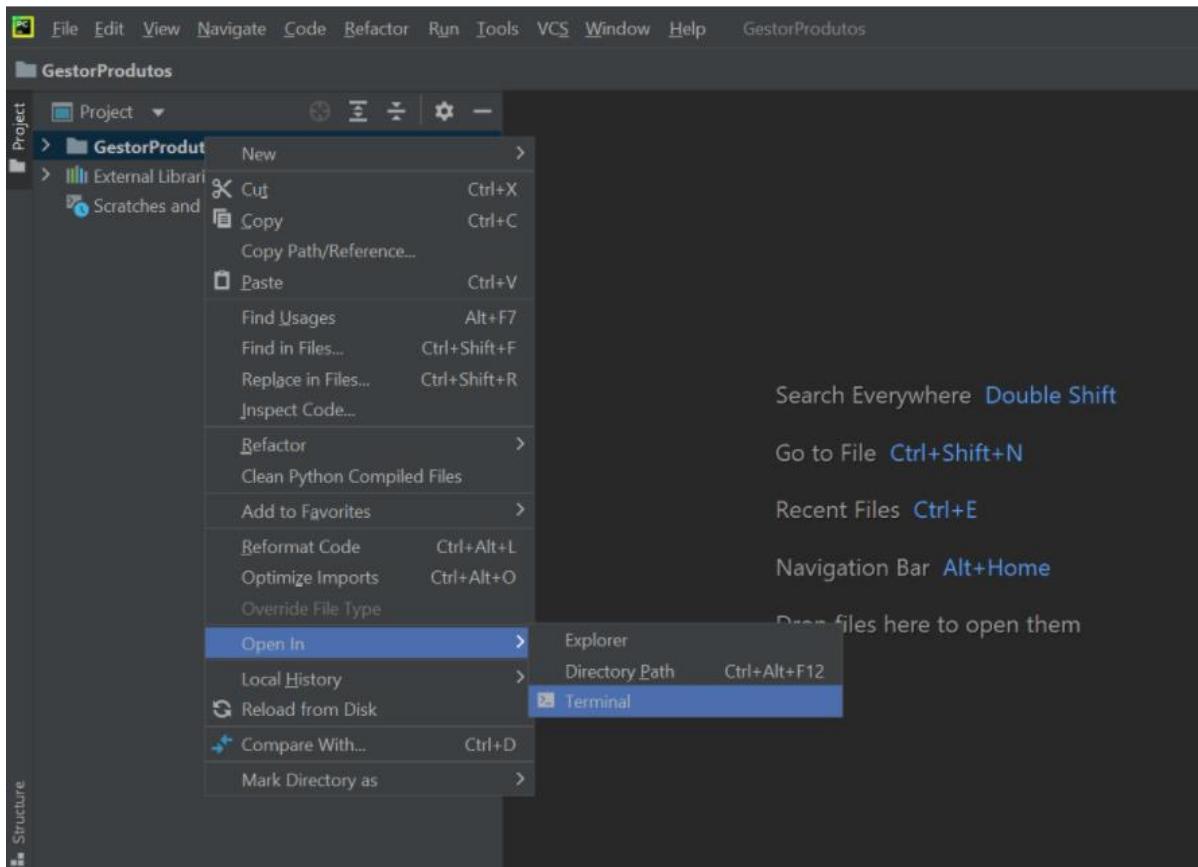
4. Abrir um terminal do projeto
  - Clicar com o botão direito sobre o projeto e em **Open in Terminal**

Nota. Para abrir um terminal em **Visual Studio Code**

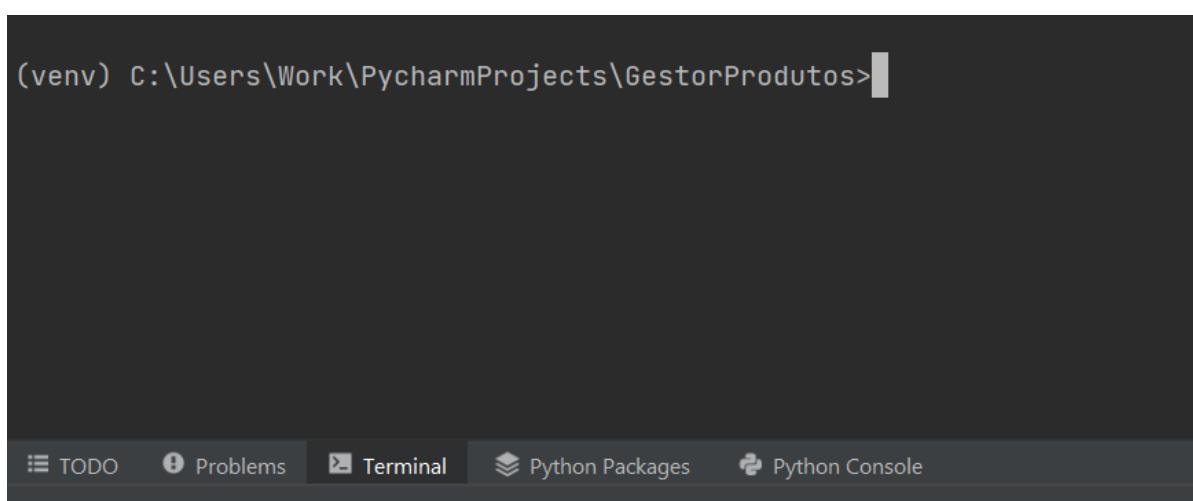
- Clicar em Ctrl + Shift + P



- Vai abrir um desdobrável de operações
- Selecionar:
  - Terminal: **Create New Integrated Terminal**



Vai abrir um **terminal** (uma consola) dentro do IDE e localizado no projeto em questão.





5. Comprovar que se tem acesso a Python desde a consola integrada do IDE:

- Verificar **Python e pip** (o instalador de módulos de Python que se vai precisar mais à frente)

```
(venv) C:\Users\Work\PycharmProjects\GestorProdutos>python --version
Python 3.9.6

(venv) C:\Users\Work\PycharmProjects\GestorProdutos>pip --version
pip 21.1.2 from C:\Users\Work\PycharmProjects\GestordeTarefas\venv\lib\site-packages\pip (python 3.9)

(venv) C:\Users\Work\PycharmProjects\GestorProdutos>
```

## 2. Instalação de módulos dentro do ambiente virtual

Antes de começar a instalar os módulos ,há que verificar que o terminal se encontra no ambiente virtual correto (fixar-se o **(venv)** no início).

```
Terminal: Local × + ▾

(venv) C:\Users\Work\PycharmProjects\GestorProdutos>
```

### IMPORTANTE

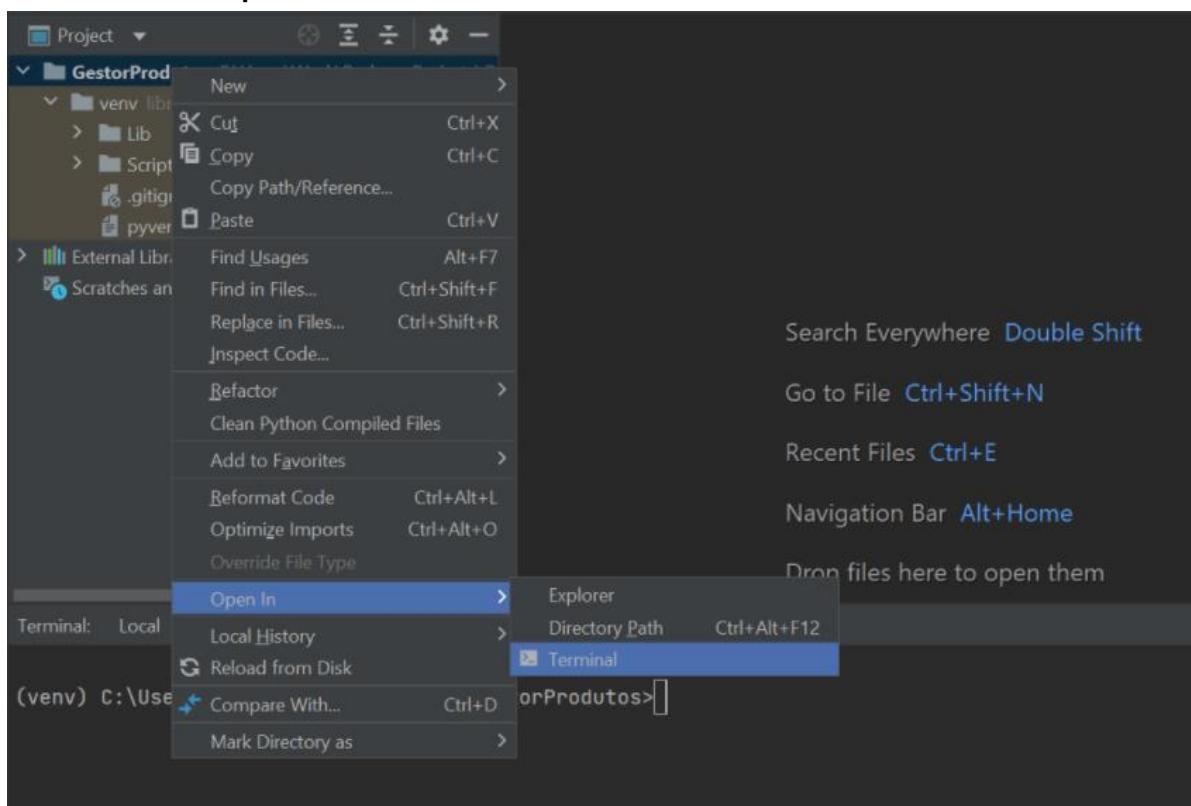


1. Neste projeto, **o módulo gráfico Tkinter já vem integrado com Python**
2. Deve instalar o módulo Flask **SQL Alchemist**, o qual vai permitir gerir o SQL a partir do servidor web Flask sem necessidade de aprofundar no idioma SQL. Para instalar este módulo, vai regressar ao terminal e executar:  
**pip install Flask-SQLAlchemy**

### 3. Sair e entrar no ambiente virtual

Se fechar o IDE, neste caso o Pycharm, também se fecha o ambiente virtual em que se está a trabalhar. Quando se voltar a abrir o Pycharm deverá voltar a entrar no ambiente virtual para continuar a trabalhar. Faz-se continuando os passos seguintes:

1. Clicar com o botão direito sobre o diretório principal do projeto
2. Clicar em **Open in Terminal**



3. Terá de aparecer um (venv) de ambiente virtual perante o Shell do projeto



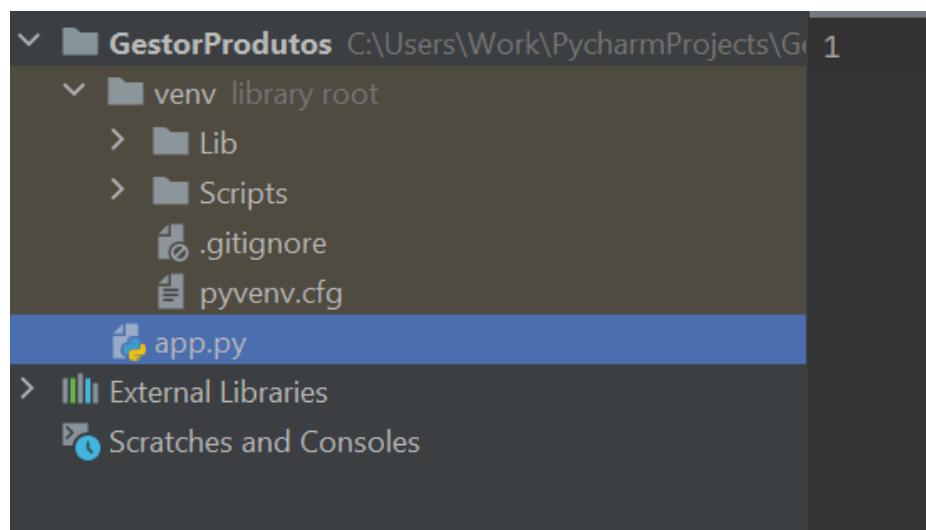
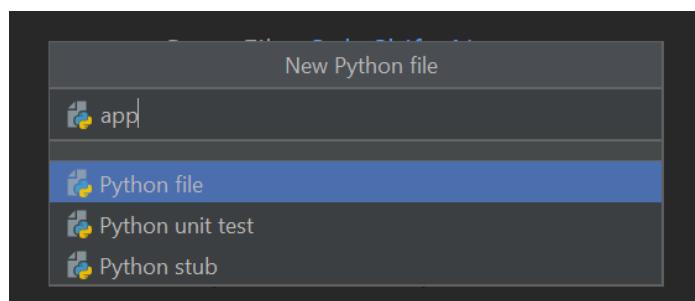
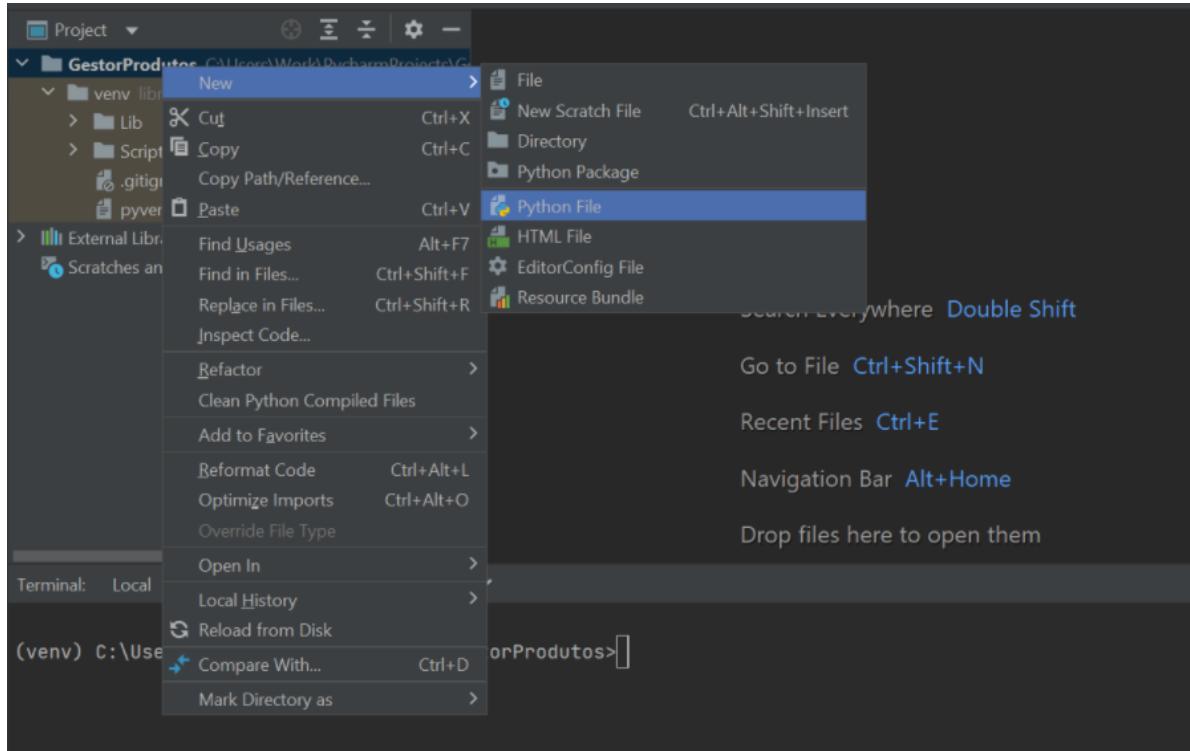
```
(venv) C:\Users\Work\PycharmProjects\GestorProdutos>
```

## 4. Criação do ficheiro Python principal e primeira janela

Embora não exista nenhuma restrição técnica na escolha de nomes dos ficheiros Python, existem certos padrões da comunidade que nos indicam os nomes padrão que se devem usar, por exemplo, em páginas web HTML, deve-se colocar **index.html** como ficheiro principal, ou em folhas de estilo, deve-se colocar **main.css**. Em Python também temos alguns nomes preferidos pela comunidade.

1. Criação do ficheiro principal de Python para este projeto: **app.py**
  - Clicar com o botão direito sobre o diretório principal do projeto
  - Clicar em **New > Python File**
  - Indicar o nome **app** e faz-se duplo clique com o botão esquerdo sobre **Python file**
  - Criar na raiz do projeto o ficheiro **app.py**

Verifique as capturas que se mostram abaixo para seguir os passos.





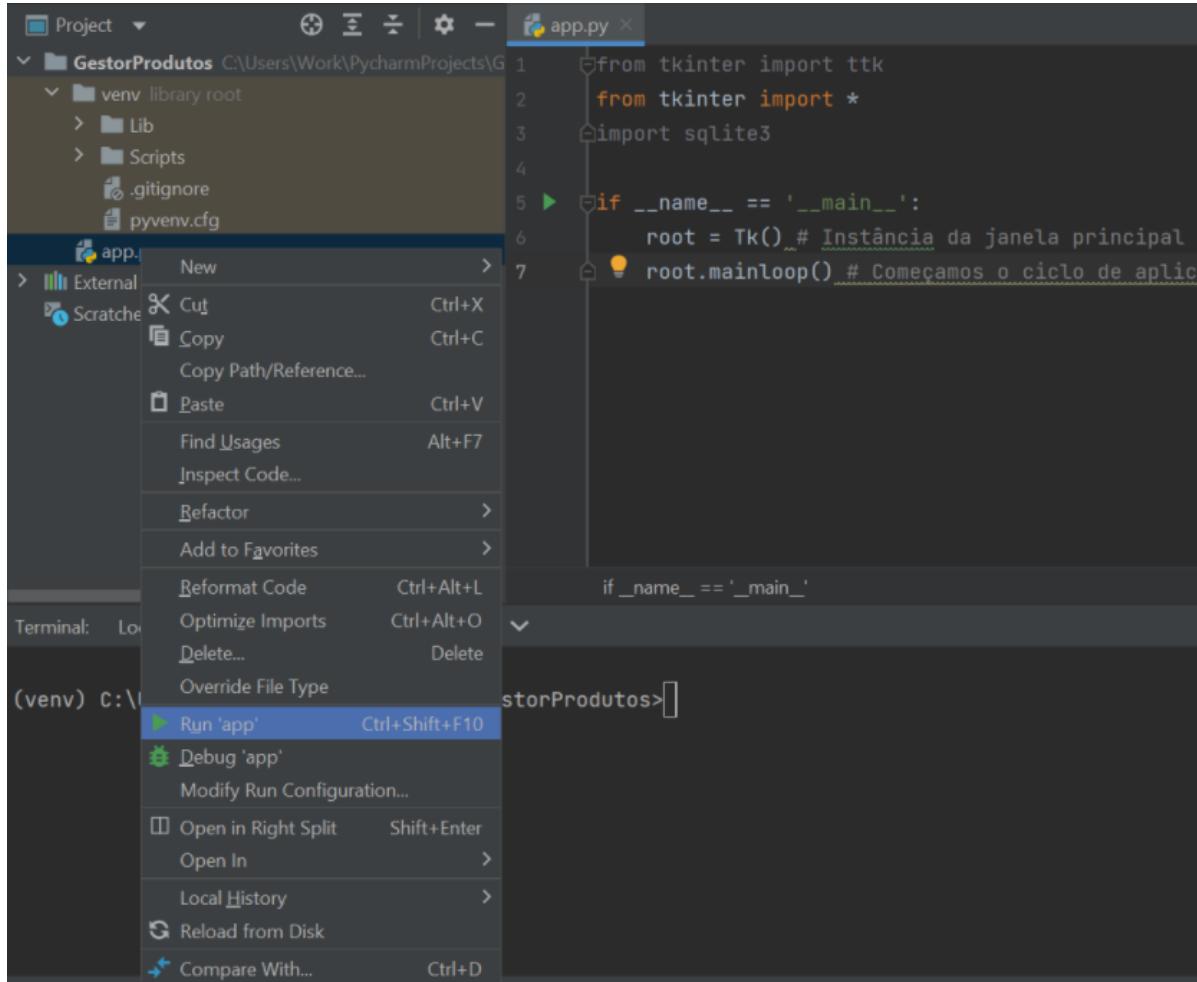
2. O primeiro objetivo, já que este é um projeto para criar uma aplicação de escritório, é criar uma janela principal (janela raiz ou **root** como chamado no Tkinter); de seguida, mostra-se o código mínimo para implementar (incluindo o *import* para a livraria Tkinter e SQLite que será usada depois):

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

if __name__ == '__main__':
    root = Tk() # Instância da janela principal
    root.mainloop() # Começamos o ciclo de aplicação, é como um while True
```

3. Executar a aplicação (duas formas):

- Através do botão **Run app** de Pycharm



- Através da consola do projeto

```
(venv) C:\Users\Work\PycharmProjects\GestorProdutos>python app.py
```

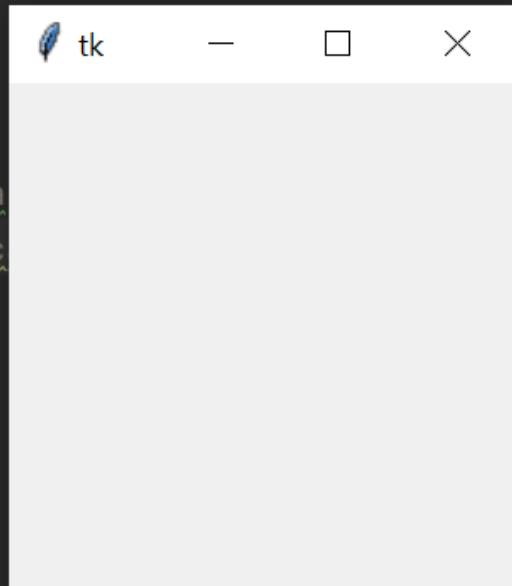
4. O resultado será uma janela vazia de Tkinter:



```
import ttk
```

```
import *
```

```
'__main__':
) # Instância
oop() # Começ
```



## 5. Começar a implementação usando Classes e Objetos



1. Podia-se programar tudo no *main*, mas quando a aplicação é de um tamanho considerável, vê-se que não é a melhor organização, pelo que se vai começar a implementar classes e objetos desde o início. Será criada uma **classe Produto**, a qual irá receber como parâmetro **o controlo da janela gráfica da aplicação** e passará a chamar-se **janela**.

```
class Produto:

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

if __name__ == '__main__':
    root = Tk() # Instância da janela principal
    app = Produto(root) # Envia-se para a classe Produto o controlo sobre a
janela root
    root.mainloop() # Começamos o ciclo de aplicação, é como um while True
```

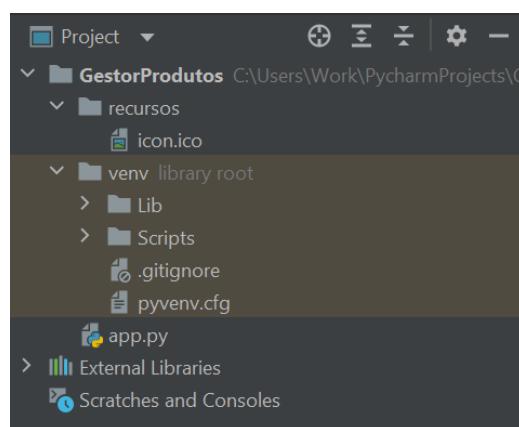
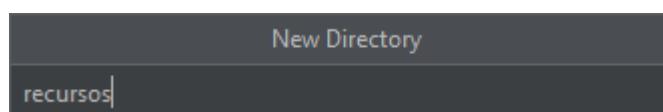
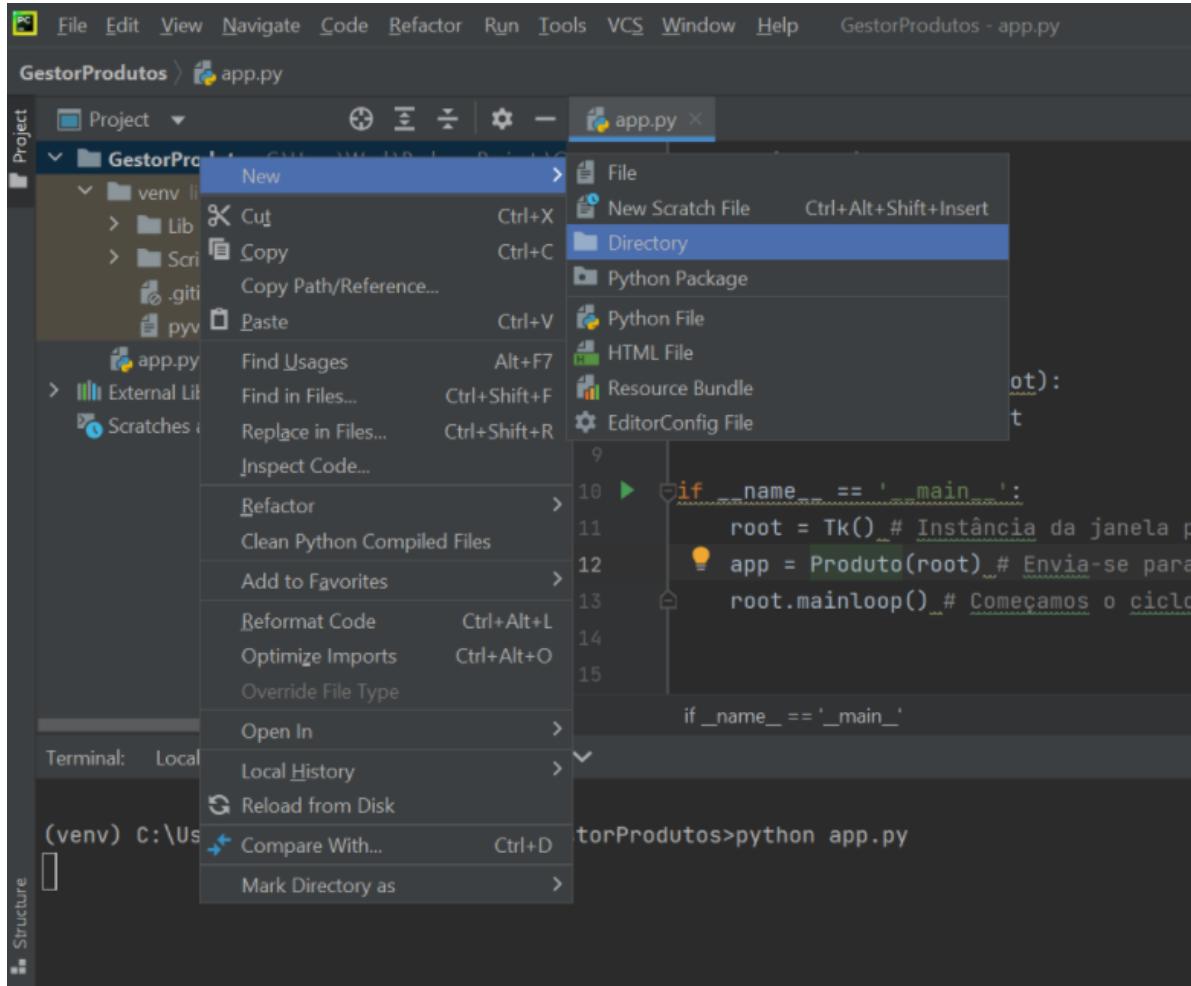
## 6. Configuração base da janela principal

1. Dentro do construtor da classe Produto tem-se a variável que controla a janela principal, chamada janela. Mas, além disso, são necessárias mais alguns detalhes:

- Um nome para a aplicação, que irá aparecer na barra de título da aplicação (por defeito é tk).
- Indicar-lhe se quer a janela redimensionada ou não.
- Um ícone (por defeito aparece o ícone de Tkinter, a pena).



2. Para poder adicionar o ícone (imagem com extensão .ico), será criada uma pasta na raiz do projeto, que será chamada de **recursos**. E vai copiar-se o ícone chamado **icon.ico** que se proporciona com este projeto para essa pasta.





3. São feitas as modificações pertinentes do código no construtor da classe **Produto**

```
class Produto:  
  
    def __init__(self, root):  
        self.janela = root  
        self.janela.title("App Gestor de Produtos") # Título da janela  
        self.janela.resizable(1,1) # Ativar a redimensionamento da janela. Para  
desativá-la: (0,0)  
        self.janela.wm_iconbitmap('recursos/icon.ico')
```

4. Experimenta-se:



## 7. Estrutura dos widgets (componentes gráficos) na janela

Antes de começar a colocar widgets (componentes gráficos) na janela, precisa entender como o Tkinter se estrutura.

Tkinter distribui a janela numa quadrícula chamada **grid**



Columnas

Filas

App Gestor de Produtos			

Esta *grid* acede-se através do seu número de fila e coluna



Columns

Filas

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

E o objetivo da *grid* é conseguir posicionar widgets no ecrã, em posições específicas. Os widgets são componentes gráficos, os quais podem ser botões, etiquetas de texto, caixas de texto para que o utilizador possa introduzir texto, imagens, sliders, etc.

Columns

Filas

Columns			
Widget 1			
Widget 2			



Para posicionar widgets individuais, a *grid* só por si iria servir, mas em muitas ocasiões necessita-se criar agrupamentos de widgets, nestes casos, cria-se outra estrutura chamada **frame**.

Os *Frames* são marcos de contentores de outros widgets (componentes gráficos). Podem ter o seu próprio tamanho e posicionar-se em lugares diferentes de outros recipientes (seja a raiz ou outro marco):



Columnas





## 8. Widgets utilizados neste projeto

**Tkinter** tem disponível uma grande quantidade de componentes gráficos em duas sub-livrarias diferentes.

- **Tk** são os componentes mais básicos e nativos (configuração muito simples)
- **Ttk** são componentes mais complexos, com uma configuração mais complicada, mas com um desenho mais cuidado.

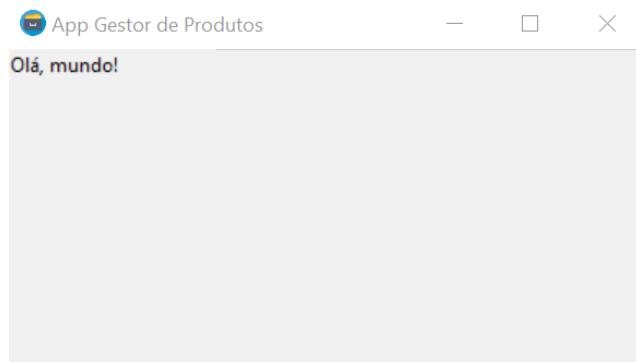
tk	ttk
Button	Button
Canvas	Checkbutton
Checkbutton	Combobox
Entry	Entry
Frame	Frame
Label	Label
LabelFrame	LabeledScale
Listbox	Labelframe
Menu	Menobutton
Menobutton	Notebook
Message	OptionMenu
OptionMenu	Panedwindow
PanedWindow	Progressbar
Radiobutton	Radiobutton
Scale	Scale
Scrollbar	Scrollbar
Spinbox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

**Na bibliografia inclui-se uma ligação para o guia não oficial de Tkinter em espanhol, principalmente para a secção widgets.**

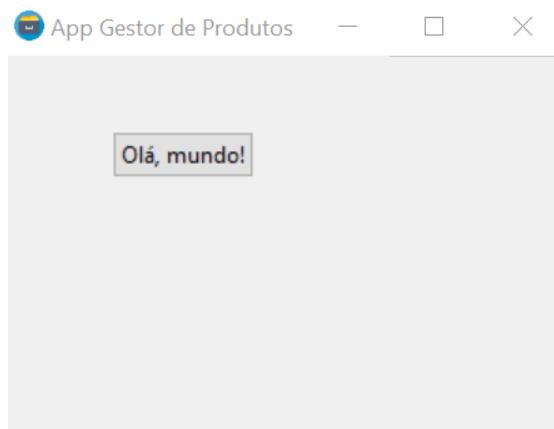


Depois, listam-se os widgets a serem usados neste projeto, com seus nomes técnicos correspondentes dentro de Tkinter:

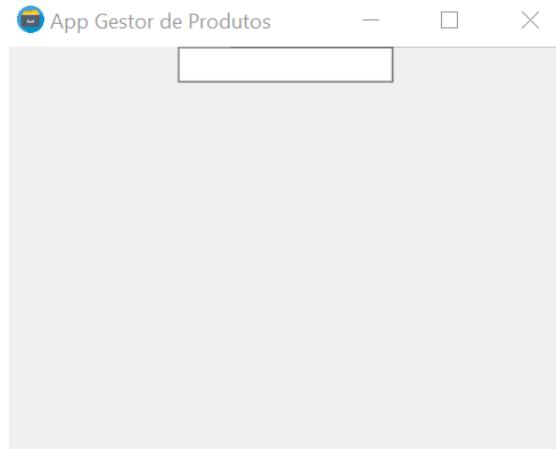
- **Label** (Etiqueta). Etiqueta de texto.



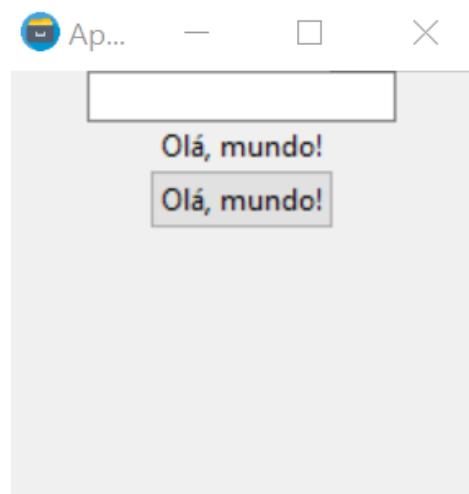
- **Button** (Botão).



- **Entry** (Quadro de texto). Serve para que o utilizador possa escrever no seu interior.



Exemplo dos 3 *widgets* anteriores juntos:





## 9. Posicionamento dos elementos em Tkinter

No Tkinter existem 3 formas de inserir *widgets* ou *frames* na janela:

1. Diretamente na **grid** (quadrículo da janela)
2. Posicionamento **absoluto**
3. Posicionamento **relativo**

Vamos verificar cada um em detalhe:

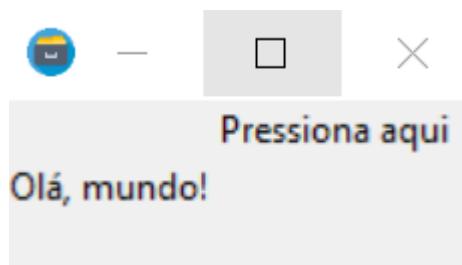
### 1. Grid (método **grid**)

Consiste em dividir a janela principal em linhas (*rows*) e colunas (*columns*), formando células onde se localizam os elementos.

Tome-se como exemplo, uma etiqueta *label* que contém o texto "Olá mundo" e que se quer colocar na linha 1, coluna 0

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

Como se vê, utiliza-se o método **grid()**, o qual tem dois parâmetros, a linha e a coluna.



O método **grid** aceita, como o **pack()**, os argumentos **padx**, **pady**, **ipadx** e **ipady** para estabelecer margens (irão ver-se no ponto 3, posicionamento relativo).

### 2. Posicionamento absoluto (método **place**)

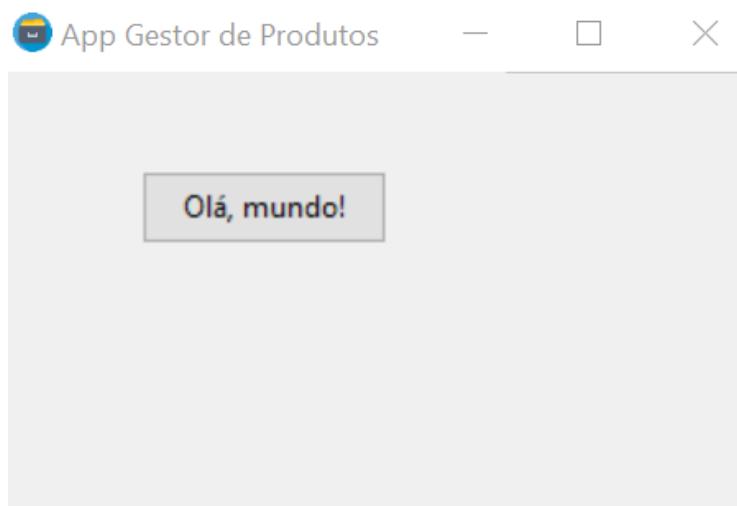
O método **place()** permite localizar elementos indicando a sua posição (X e Y) em relação a um elemento pai. No geral, quase todas as livrarias gráficas fornecem tal opção, pois é a mais intuitiva.

Como exemplo, um botão que deseja colocar na posição (60, 40)



```
self.button = ttk.Button( text="Olá, mundo!")  
self.button.place(x=60, y=40)
```

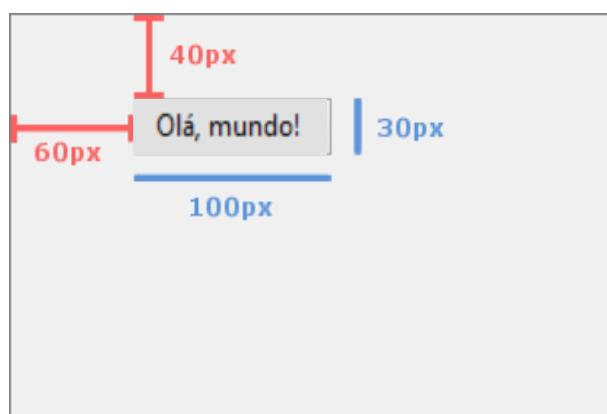
Já que a origem das coordenadas (ou seja, a posição (0, 0)) é o canto superior esquerdo, significa que entre a borda esquerda da janela e o nosso botão haverá uma distância de 60 pixéis e entre a borda superior da janela e o botão será de 40 pixéis.



É possível indicar o tamanho de qualquer outro elemento Tkinter utilizando os parâmetros de **width** e **height**, que indicam a largura e altura em pixéis.

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

A imagem seguinte ilustra como os quatro argumentos (**x**, **y**, **width**, **height**) influenciam a posição e o tamanho do *widget*.





### 3. Posicionamento relativo (método *pack*)

Este método é o mais simples dos três. Em vez de especificar as coordenadas de um elemento, simplesmente dizemos para deve ir acima, abaixo, para a esquerda ou para a direita em relação a algum outro *widget* ou à janela principal.

Como por exemplo, o desenho de uma **Entry** (caixa de texto), um botão e uma **label** (etiqueta de texto). Para localizar os elementos vamos usar o método **pack()** e se o usamos sem argumentos, por defeito, colocará os elementos sobrepostos.

```
self.entry = ttk.Entry()  
self.entry.pack()  
  
self.button = ttk.Button( text="Olá, mundo!")  
self.button.pack()  
  
self.label = ttk.Label( text="...desde Tkinter!")  
self.label.pack()
```

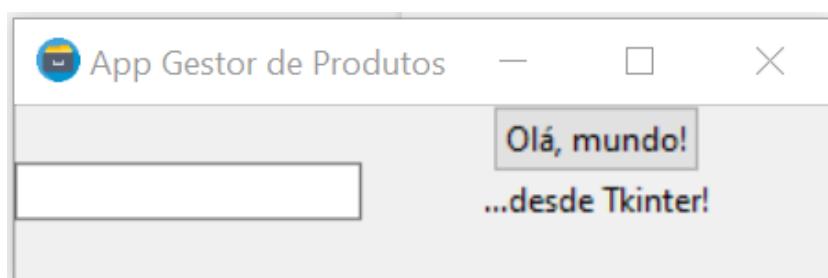


Mas o **pack()** é muito mais poderoso do que isto e proporciona-nos a capacidade de colocar os elementos onde se queira, graças à **propriedade side**, a qual controla a posição relativa dos elementos:

- **Tk.TOP** (por defeito)
- **Tk.BOTTOM**
- **Tk.LEFT**
- **Tk.RIGHT**

Deste modo, se indicarmos que a caixa de texto deve estar localizada à esquerda, os outros dois *widgets* vão continuar a estar sobrepostos.

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



Outras propriedades interessantes são **after** e **before** para indicar que um *widget* deve ir antes ou depois do outro.



E outras propriedades são **padx**, **ipadx**, **pady** e **ipady** que especificam (em pixéis) as margens externas e internas de um elemento. Por exemplo, no código seguinte haverá um espaço de 30 pixéis entre o botão e a janela (margem externa), mas um espaço de 50 pixéis entre a borda do botão e o texto do mesmo (margem interna).

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



Finalmente, é possível especificar que elementos devem expandir-se ou contrair-se à medida que o tamanho da janela muda e em que sentido (vertical ou horizontal), através das propriedades de **expand** e **fill**.

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

Neste exemplo, o botão só vai ajustar o seu tamanho horizontal (**fill=tk.X**).

Para aumentar a janela podemos fazer:

```
root.geometry("500x100") # Width x Height
```



## 10. Começar com a interface gráfica Adicionar produto

1. Criação de um **frame** principal que engloba todos os widgets relacionados à funcionalidade de adicionar produtos. No construtor da classe Produto, adiciona-se o seguinte:

```
# Criação do recipiente Frame principal
frame = LabelFrame(self.janela, text = "Registrar um novo Produto")
frame.grid(row = 0, column = 0, columnspan = 3, pady = 20)
```

**LabelFrame** é um widget filho de *Frame*. Trata-se de um recipiente de *widgets* como o pai. A diferença é que desenha um limite à volta do seu tamanho.

Este *frame* começa na **posição 0.0** da janela, e com **columnspan** indica quantas colunas da *grid* vai usar. Também se inclui uma pequena margem no eixo e a instrução **pady**.



## 2. Criação da *Label* e *Entry* de Nome

Continuamos no construtor da classe Produto e adiciona-se o seguinte:

```
# Label Nome
self.etiqueta_nome = Label(frame, text="Nome: ") # Etiqueta de texto localizada
no frame
self.etiqueta_nome.grid(row=1, column=0) # Posicionamento através de grid
# Entry Nome (caixa de texto que irá receber o nome)
self.nome = Entry(frame) # Caixa de texto (input de texto) localizada no frame
self.nome.focus() # Para que o foco do rato vá a esta Entry no início
self.nome.grid(row=1, column=1)
```

Como se observa, cria-se a **Label** de nome, indicando que vai existir dentro do *frame*. E coloca-se com o método de **grid()** na linha 1 e coluna 0. Depois cria-se **Entry** de nome, a caixa de texto onde os utilizadores irão escrever o nome. Indica-se também que existe dentro do *frame* e que está localizado através do método **grid()** na linha 1 e coluna 1 (ou seja, à direita da sua *Label*).

Por último, também se utiliza o método **focus()** para centrar a seta do rato na *Entry* de Nome, ou seja, quando se executar a aplicação, a *Entry* de Nome seja diretamente acessível.

## 3. Criação da Label e a Entry de preço

```
# Label Preço
self.etiqueta_preço = Label(frame, text="Preço: ") # Etiqueta de texto
localizada no frame
self.etiqueta_preço.grid(row=2, column=0)
# Entry Preço (caixa de texto que irá receber o preço)
self.preco = Entry(frame) # Caixa de texto (input de texto) localizada no frame
self.preco.grid(row=2, column=1)
```

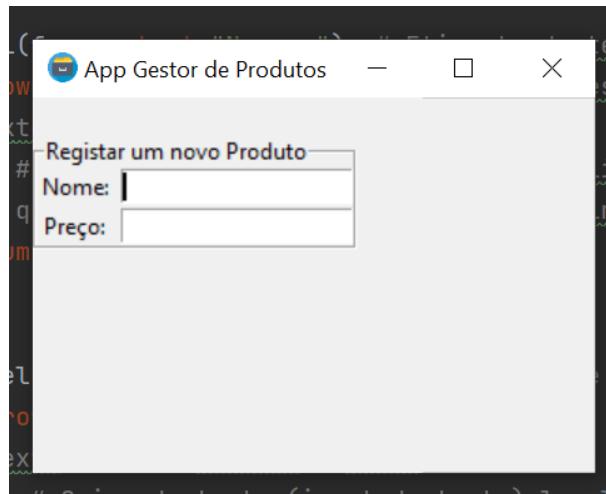
## 4. Comprovação do implementado



Se nas propriedades do construtor indicarmos que a janela pode ser redimensionada (`self.janela.resizable(1,1)`) pode-se fazer a janela maior para ver todos os detalhes da implementação, como margens, etc.

Podemos também modificar a geometry:

```
root.geometry("300x200") # Width x Height
```



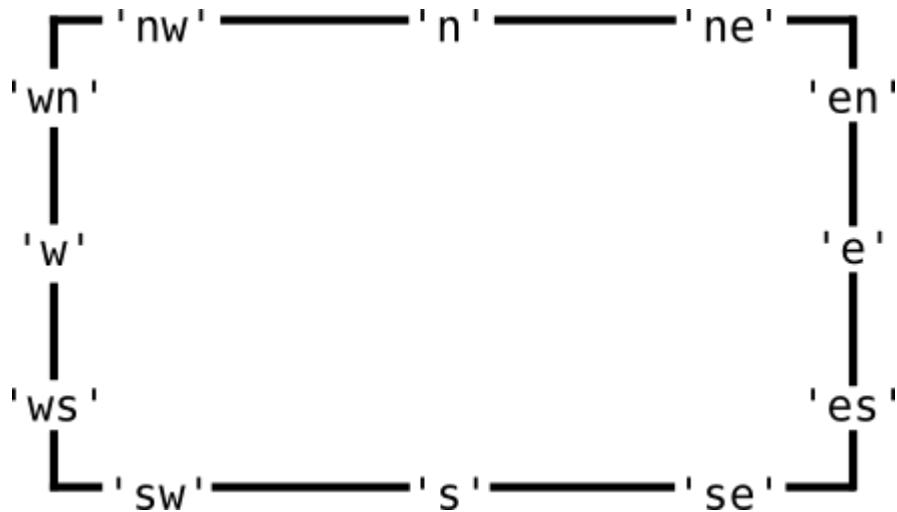
## 5. Criação do botão Guardar produto

```
# Botão Adicionar Produto
self.botao_adicionar = ttk.Button(frame, text = "Guardar Produto")
self.botao_adicionar.grid(row = 3, columnspan = 2, sticky = W + E)
```

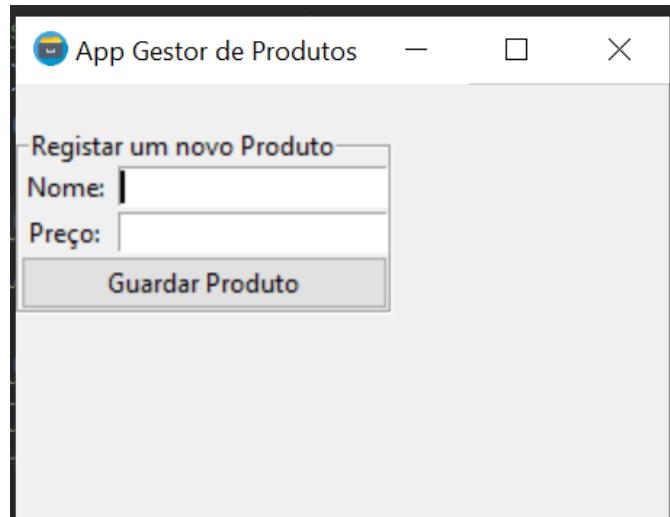
Neste caso, evita-se usar o Ñ no código para evitar problemas sintáticos. Este código cria um botão com o texto *Guardar produto*, dentro do *frame*, localizado na linha 3 e, com o atributo **sticky**, diz-se quanto se quer que ocupe. Neste caso é dito para ocupar toda a largura, do Oeste (W) ao Leste (E).



Estas etiquetas de posicionamento pertencem ao *frame* e de seguida podem ver-se todas as que estão disponíveis.



## 6. Comprovação do implementado





## 11. Interface gráfica. Tabela de produtos

Neste ponto, há um formulário para que o utilizador possa guardar novos produtos. Mas, desde a própria janela da aplicação, o utilizador terá de ver os produtos existentes. Para isso, vai-se criar uma tabela (*widget Treeview da sub-livraria ttk*).

1. Criação da **tabela de Produtos**. Irá criar-se um estilo próprio para a tabela, modificando o texto dos cabeçalhos, tornando-os maiores e que estejam a negrito, eliminando as bordas da tabela, etc. As fontes e tamanhos podem ser trocados livremente.

Continuamos no construtor da classe **Produto**:

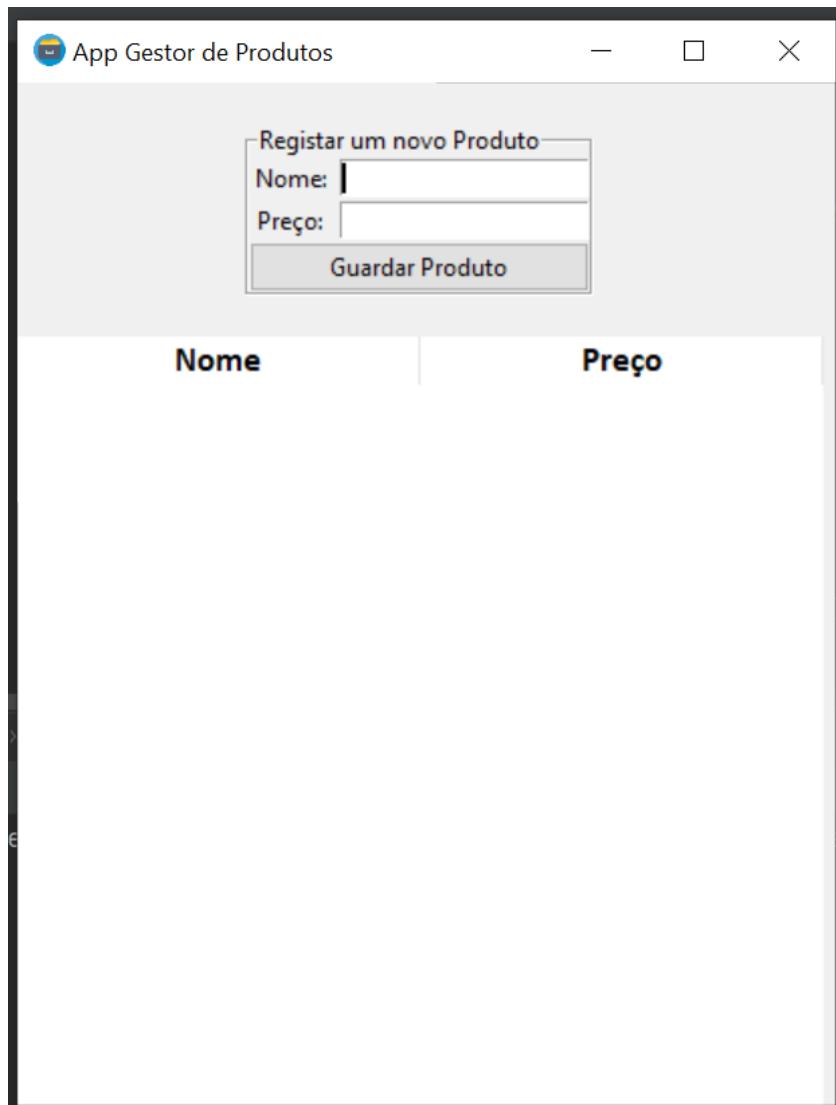
```
# Tabela de Produtos
# Estilo personalizado para a tabela
style = ttk.Style()
style.configure("mystyle.Treeview", highlightthickness=0, bd=0, font=('Calibri', 11)) # Modifica-se a fonte da tabela
style.configure("mystyle.Treeview.Heading", font=('Calibri', 13, 'bold')) # Modifica-se a fonte das cabeceiras
style.layout("mystyle.Treeview", [('mystyle.Treeview.treearea', {'sticky': 'nswe'})]) # Eliminar as bordas

# Estrutura da tabela
self.tabela = ttk.Treeview(height = 20, columns = 2, style="mystyle.Treeview")
self.tabela.grid(row = 4, column = 0, columnspan = 2)
self.tabela.heading('#0', text = 'Nome', anchor = CENTER) # Cabeçalho 0
self.tabela.heading('#1', text='Preço', anchor = CENTER) # Cabeçalho 1
```

Na parte de construção da estrutura diz que tem uma altura de 20 linhas e uma largura de 2 colunas. Pode-se variar a seu gosto. Localiza-se com o método de **grid()** na linha 4 (falta localizar um elemento na linha 3 mais à frente). Por último, adicionam-se os cabeçalhos da tabela.



## 2. Comprovação do implementado



Agora seria o momento de carregar o conteúdo da tabela com os produtos. Para isso são necessários dados, armazenados numa base de dados.



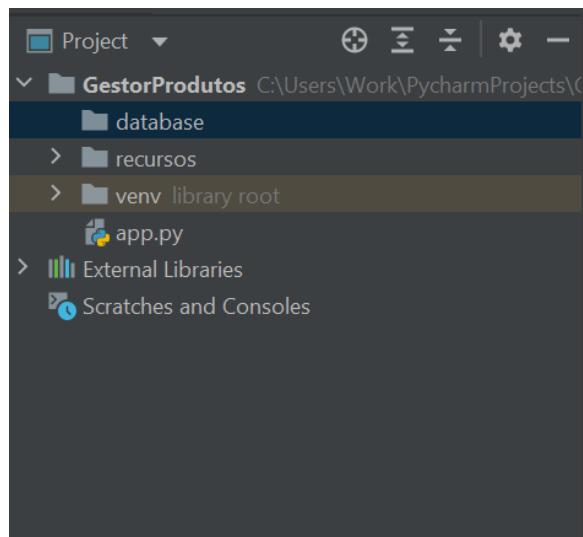
## 12. Criação da base de dados

Este projeto irá utilizar uma base de dados **SQLite** e o gestor **DB Browser (for SQLite)**. Fica fora do âmbito deste projeto a instalação de ambos, portanto, se neste ponto não estão instalados no computador, terá de ser feito imediatamente para poder continuar. As ligações às páginas de transferências encontram-se na bibliografia deste projeto.

A criação e configuração da base de dados que se detalha a seguir poderia ser feita através do código, mas para variar de outros projetos, vai-se usar o **programa DB Browser** para o fazer.

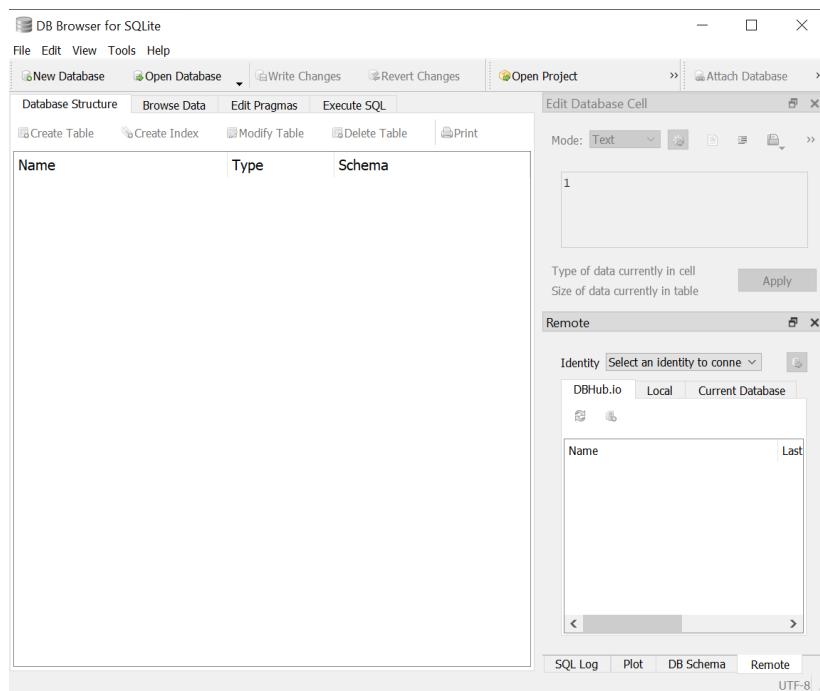
### 1. Criação da pasta onde será armazenada a base de dados

Assim como se fez com a pasta de recursos, vai-se criar uma pasta chamada **database**





## 2. Será executado **DB Browser (for SQLite)**



## 3. Deve ir a uma **Nova base de dados** na parte superior esquerda.

Irá abrir-se uma janela pop-up para indicar ao programa onde quer criar a base de dados.

Deverá dirigir-se à pasta que acabou de criar:

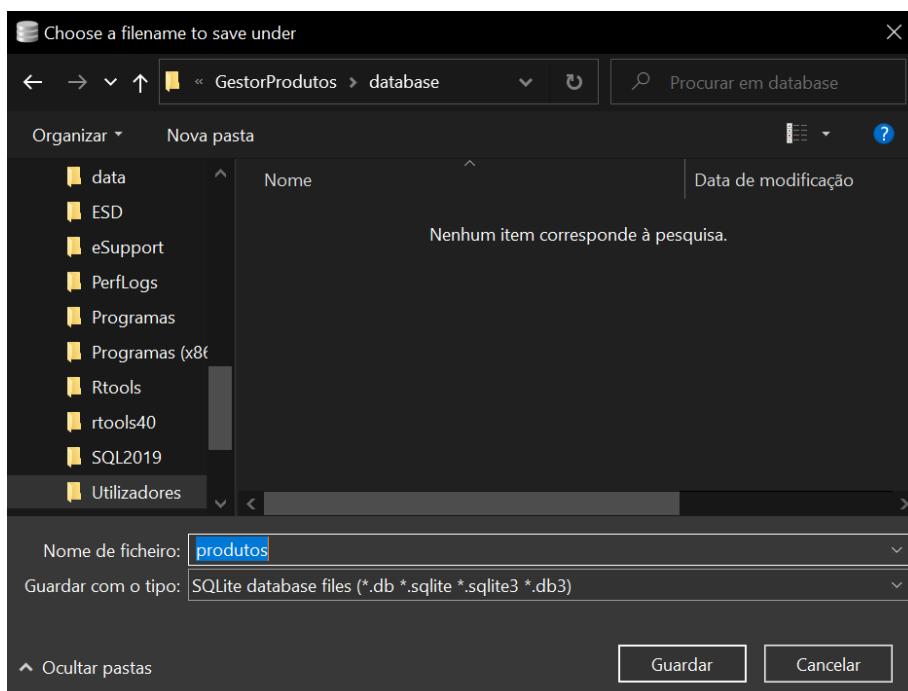
**PycharmProjects\GestorProdutos\database.**

No nosso caso específico, a rota absoluta é:

**C:\Users\<Utilizador>\PycharmProjects\GestorProdutos\database**, mas esta rota irá depender do nome do utilizador do sistema e da localização do *workspace* do IDE de desenvolvimento que se utilize.



Quando a localização estiver correta, será dado o nome **produtos** à base de dados e clica-se em Guardar.



4. Depois, é aberta uma janela onde o DB Browser pergunta diretamente ao utilizador se deseja criar uma tabela para a sua base de dados. Vamos criar a tabela **produto**. Nas bases de dados, devemos nomear nomes da base de dados no plural e das tabelas no singular. É por isso que esta base de dados se chama produtos e a tabela produto.  
Esta **tabela produtos terá 3 campos** (para adicionar os campos tem de clicar em **Adicionar campo**) com os seguintes atributos:
  - o **id**
    - Tipo: **INTEGER** (número inteiro)



- NN: **Not null** (não pode ser nulo, não se pode deixar este campo vazio)
  - PK: **Primary Key** (chave primária, campo único que não irá permitir repetições e irá servir para identificar inequivocamente a cada linha de dados)
  - AI: **AutoIncrement** (número auto-incremental). O gestor da base de dados encarrega-se de gerar automaticamente este número 1, 2, 3, ... sem repetições.
- **nome**
    - Tipo: **TEXT**
    - NN: **Not null** (não pode ser nulo, não se pode deixar este campo vazio)
  - **preço**
    - Tipo: **REAL** (número real, com decimais)
    - NN: **Not null** (não pode ser nulo, não se pode deixar este campo vazio)

O resultado da configuração deve ser o seguinte:

The screenshot shows the 'Edit table definition' dialog box. In the 'Table' section, 'produto' is selected. Under the 'Fields' tab, there are three fields: 'id' (Type: INTEGER, NN: checked, PK: checked, AI: checked), 'nome' (Type: TEXT, NN: checked), and 'preco' (Type: REAL, NN: checked). In the 'Constraints' tab, the primary key constraint is defined as PRIMARY KEY('id' AUTOINCREMENT). The generated SQL code at the bottom is:

```
1 CREATE TABLE "produto" (
2     "id"      INTEGER NOT NULL,
3     "nome"    TEXT NOT NULL,
4     "preco"   REAL NOT NULL,
5     PRIMARY KEY("id" AUTOINCREMENT)
6 );
```



Ao finalizar a configuração clicar em Aceitar para criar a tabela. Irá voltar à janela principal do **DB Browser**, onde se poderá ver a tabela recém-criada

The screenshot shows the DB Browser for SQLite interface. The main window title is "DB Browser for SQLite - C:\Users\Work\PycharmProjects\GestorProdutos\database\produtos.db". The menu bar includes File, Edit, View, Tools, Help, and several database-related buttons like New Database, Open Database, Write Changes, Revert Changes, Open Project, Save Project, and Attach Database.

The "Database Structure" tab is selected. In the left pane, under "Tables (2)", the "produto" table is expanded, showing its three columns: "id" (Type: INTEGER), "nome" (Type: TEXT), and "preco" (Type: REAL). To the right of the table definition, the schema is displayed:

```
CREATE TABLE "produto" ( "id" INTEGER NOT NULL, "id" INTEGER NOT NULL, "nome" TEXT NOT NULL, "preco" REAL NOT NULL)
```

Below the table definitions, there are sections for Indices (0), Views (0), and Triggers (0).

In the center-right area, there is an "Edit Database Cell" dialog open. It shows a single cell containing the value "1". Below the cell, it says "Type of data currently in cell" and "Size of data currently in table". There is a "Mode: Text" dropdown and a toolbar with icons for copy, paste, and save.

At the bottom right of the interface, there is a "Remote" panel with tabs for Identity (set to "DBHub.io"), Local, and Current Database. It also shows a "Name" field with "Last" and navigation arrows.

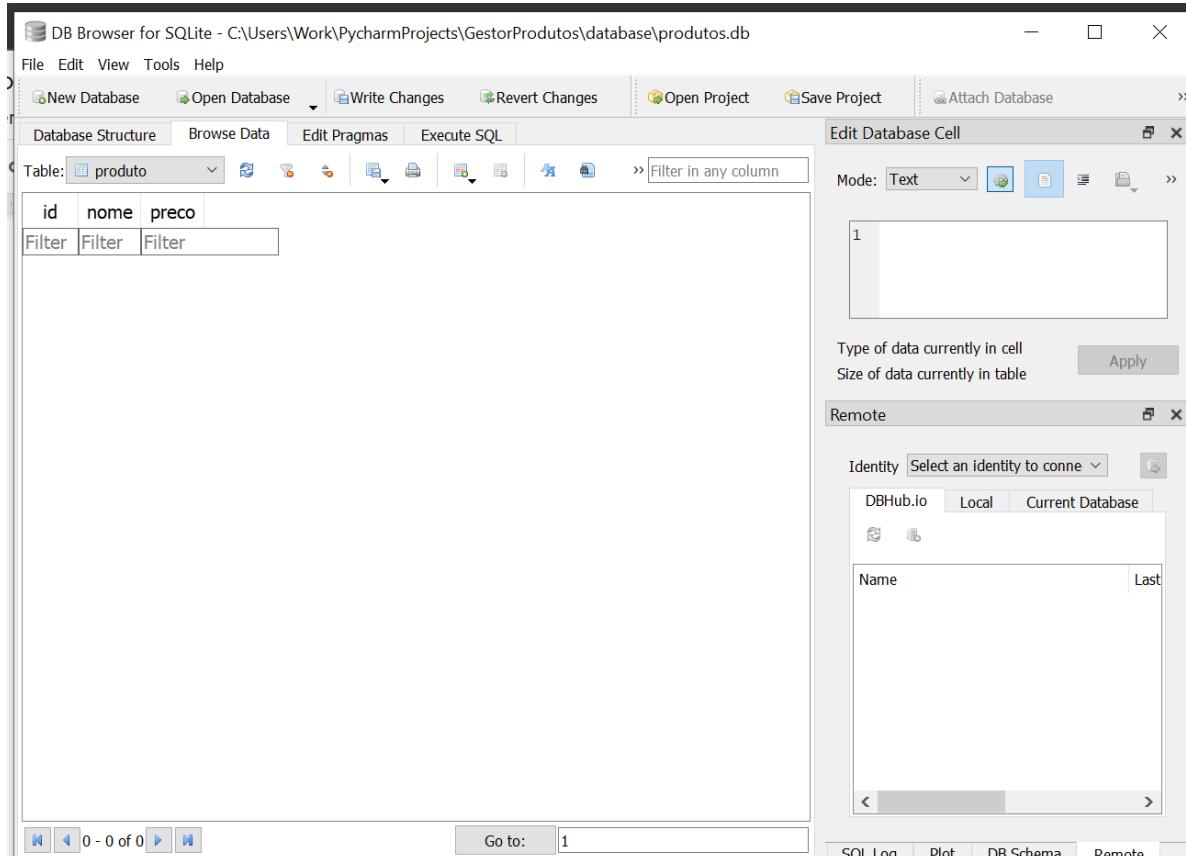
At the very bottom of the interface, there are tabs for SQL Log, Plot, DB Schema, Remote, and a language setting of UTF-8.



### 13. Inserir dados de teste na base de dados

Vamos inserir alguns registros (umas linhas) de dados de amostra para poder aceder desde a aplicação, antes de implementar a funcionalidade de guardar novos produtos.

1. Para isso, deve ir ao separador **Folha de dados** do **DB Browser**



2. Os seguintes regtos serão inseridos. Há que ter em conta que o id do produto não precisa ser introduzido manualmente, pois configurou-se como auto-incremental, portanto o gestor irá encarregar-se de o adicionar.
  - id 1
    - nome: **Impressora 3D**
    - preço: **1000**
  - id 2
    - nome: **Arduino**
    - preço: **20**
  - id 3
    - nome: **Raspberry Pi 4**
    - preço: **50**
3. Para criar os regtos, clicar em **Novo Registo**



Database Structure		Browse Data	Edit Pragmas	Execute SQL
Table: <b>produto</b>				
		id	nome	preco
		id1	impressora 3D	1000

4. Depois de clicar em Novo registo comprova-se se o id é inserido automaticamente. Para inserir os dados desejados apenas precisa clicar no nome ou preço e preencher com os dados no ecrã da direita. Quando terminar, clicar em **Aplicar**

The screenshot shows the SQLite Manager interface. On the left, a table named 'produto' is displayed with three columns: 'id', 'nome', and 'preco'. The first row has an 'id' of 1, a 'nome' of 'Impressora\_3D', and a 'preco' of 1000. On the right, an 'Edit Database Cell' dialog is open over the 'nome' cell of the first row. The input field contains the text 'Impressora\_3D'. Below the input field, it says 'Type of data currently in cell: Text / Numeric 13 character(s)'. There is also an 'Apply' button at the bottom right of the dialog.

5. Quando terminar de inserir os registos, deve-se clicar em **Guardar alterações**

Write Changes				
Database Structure		Browse Data	Edit Pragmas	Execute SQL
Table: <b>produto</b>				
id			nome	preco
Filter	Filter	Filter		
1	1	Impressora_3D		1000
2	2	Arduino		20
3	3	Raspberry_Pi		50

Quando os dados estejam guardados, o botão de Guardar alterações será desativado:

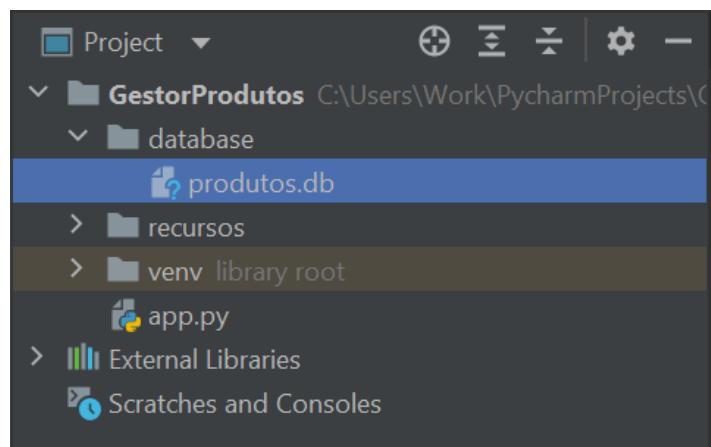




## 14. Implementar a conexão à base de dados a partir do Python

Neste ponto, está criada a base de dados com uma tabela e vários registo inseridos. De seguida, vai-se programar a conexão à base de dados para poder aceder ao seu conteúdo.

1. Verificar que se vê a base de dados desde Pycharm na estrutura de pastas e ficheiros.



2. Verifique se tem o *import* para **sqlite** corretamente adicionado no início do programa

```
import sqlite3
```

3. Criar uma variável na classe **Produto** para aceder à rota da base de dados.

```
class Produto:  
    db = 'database/produtos.db'
```

4. Criar um método na classe **Produto** que se conecte à base de dados, execute uma consulta e feche a conexão.



```
def db_consulta(self, consulta, parametros = ()):
    with sqlite3.connect(self.db) as con: # Iniciamos uma conexão com a base de
dados (alias con)
        cursor = con.cursor() # Criamos um cursor da conexão para poder operar na
base de dados
        resultado = cursor.execute(consulta, parametros) # Preparar a consulta
SQL (com parâmetros se os há)
        con.commit() # Executar a consulta SQL preparada anteriormente
    return resultado # Restituir o resultado da consulta SQL
```



## 15. Implementar o método de get\_produtos

1. Criar um método na classe **Produto** que devolva uma listagem de todos os produtos na base de dados.

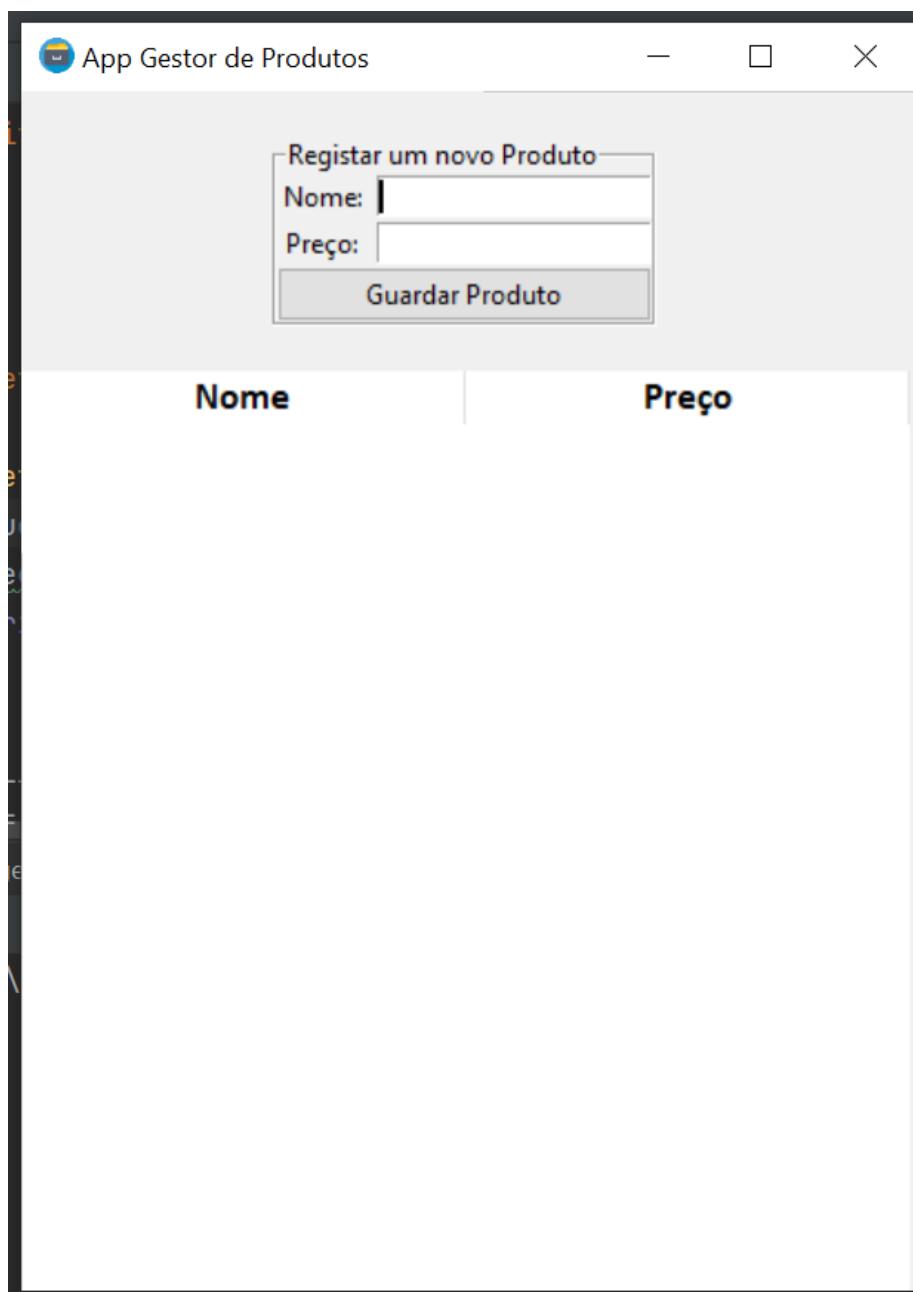
```
def get_produtos(self):
    query = 'SELECT * FROM produto ORDER BY nome DESC'
    registos = self.db_consulta(query) # Faz-se a chamada ao método db_consultas
    print(registos) # Mostram-se os resultados
```

2. Adicionar no final do construtor da classe **Produto** a chamada ao método **get\_produtos()** o qual nos irá devolver a listagem de produtos ao início da aplicação.

```
# Chamada ao método get_produtos() para obter a listagem de produtos ao início da app
self.get_produtos()
```



3. Experimentar o implementado. Executar a aplicação





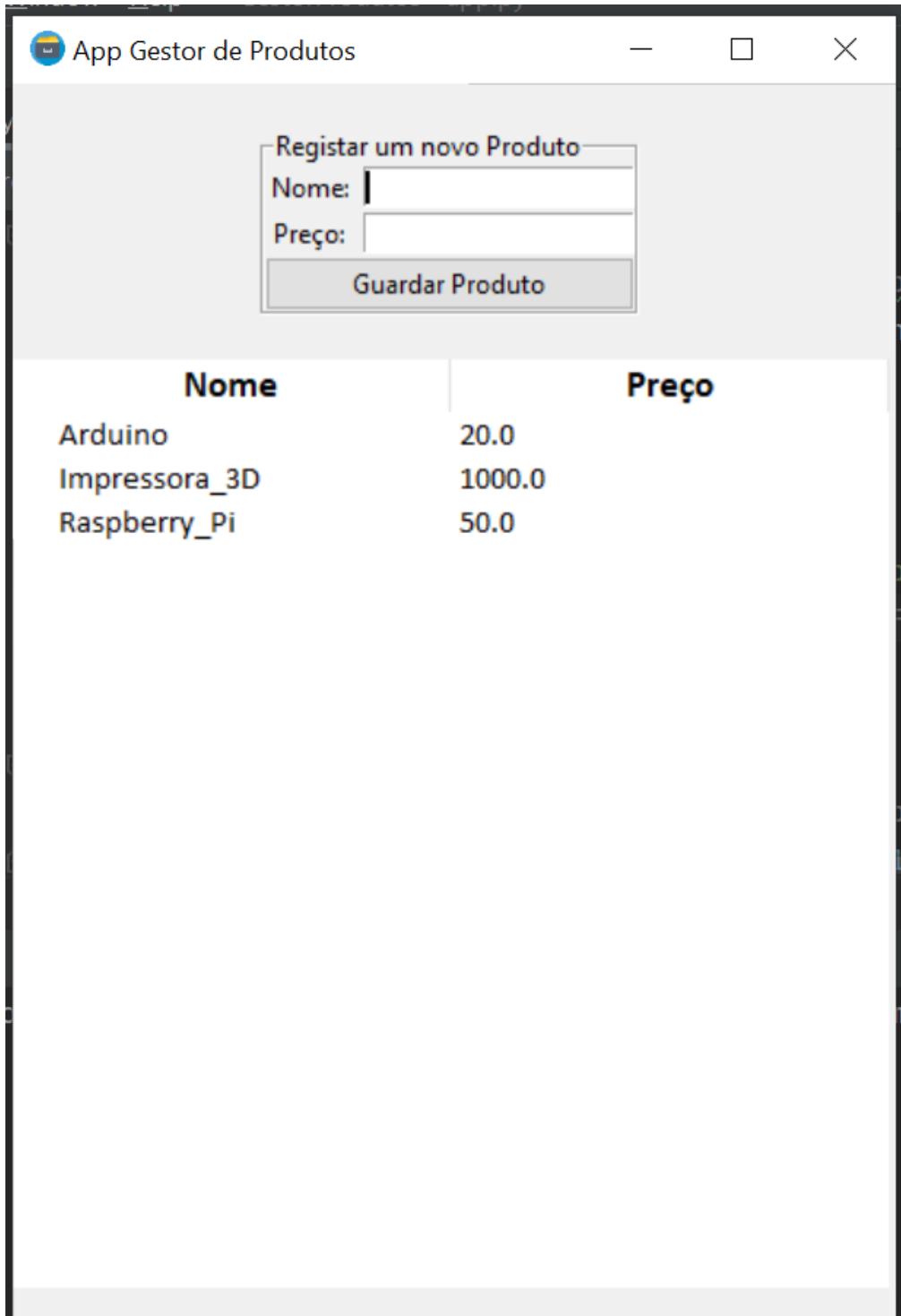
Verificar se as informações devolvidas são o cursor da base de dados, não os dados em si.

4. Vai-se **modificar get\_produtos()** para poder devolver os dados e não o cursor. Para isso, o método será dividido em 3 partes:
  - a. Limpar os dados que a tabela possa ter
  - b. Fazer a consulta SQL
  - c. Inserir o resultado da consulta na tabela
    - i. Além disso, para poder verificar os resultados, está incluído um print que mostra os produtos por consola.

```
def get_produtos(self):  
    # O primeiro, ao iniciar a app, vamos limpar a tabela se tiver dados  
    # residuais ou antigos  
    registos_tabela = self.tabela.get_children() # Obter todos os dados da tabela  
    for linha in registos_tabela:  
        self.tabela.delete(linha)  
  
    # Consulta SQL  
    query = 'SELECT * FROM produto ORDER BY nome DESC'  
    registos_db = self.db_consulta(query) # Faz-se a chamada ao método  
    db_consultas  
  
    # Escrever os dados no ecrã  
    for linha in registos_db:  
        print(linha) # print para verificar por consola os dados  
        self.tabela.insert('', 0, text = linha[1], values = linha[2])
```



5. Experimentar o implementado. Executar a aplicação



```
app.x
C:\Users\Work\PycharmProjects\GestorProdutos\venv\Scripts\python.exe C:/Users/Work/PycharmProjects/GestorProdutos/app.py
(3, 'Raspberry_Pi', 50.0)
(1, 'Impressora_3D', 1000.0)
(2, 'Arduino', 20.0)
```



## 16. Implementar o método de add\_produto() e as suas validações

1. Criar um método na classe **Produto** que crie um novo produto e o insira na base de dados. Este método será chamado de **add\_produto()**, mas irá exigir outros métodos secundários para validar os dados que o utilizador introduziu através do formulário. Estes métodos serão chamados **validação\_nome()** e **validação\_preco()**

```
def validacao_nome(self):
    nome_introduzido_por_utilizador = self.nome.get()
    return len(nome_introduzido_por_utilizador) != 0

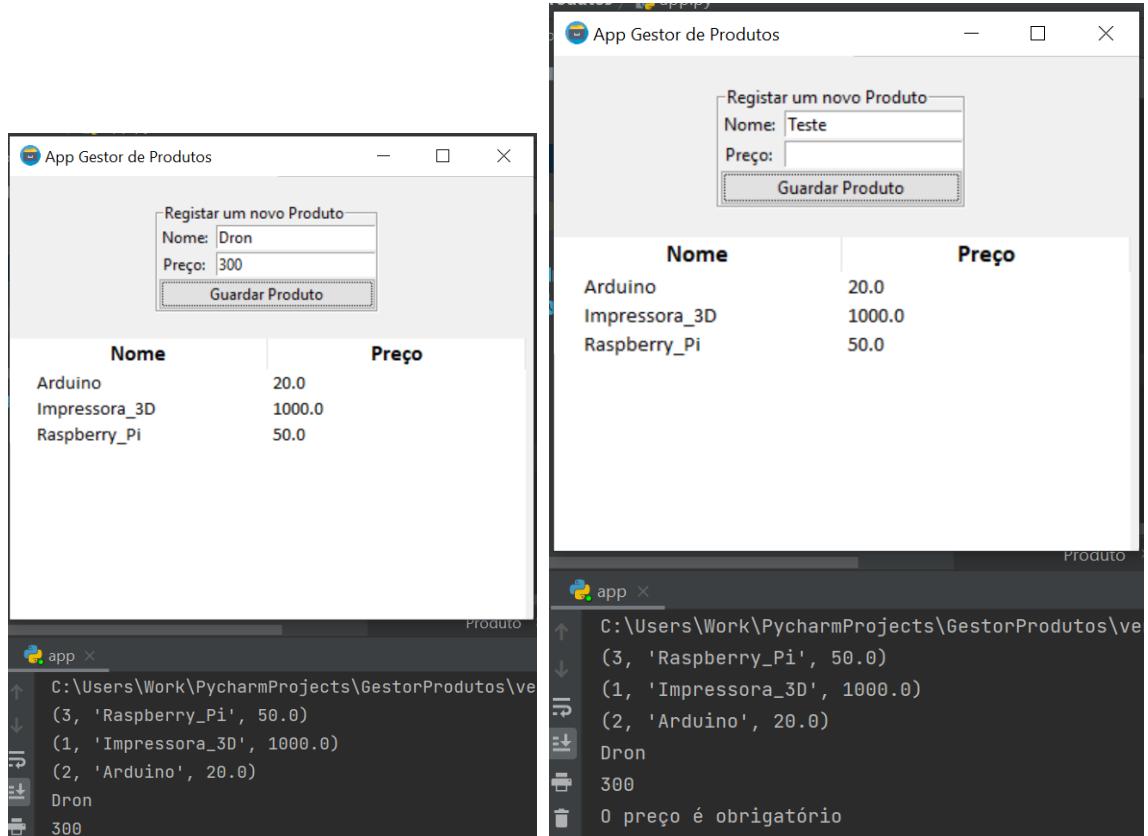
def validacao_preco(self):
    preco_introduzido_por_utilizador = self.preco.get()
    return len(preco_introduzido_por_utilizador) != 0

def add_produto(self):
    if self.validacao_nome() and self.validacao_preco():
        print(self.nome.get())
        print(self.preco.get())
    elif self.validacao_nome() and self.validacao_preco() == False:
        print("O preço é obrigatório")
    elif self.validacao_nome() == False and self.validacao_preco():
        print("O nome é obrigatório")
    else:
        print("O nome e o preço são obrigatórios")
```

2. Antes de testar o código implementado anteriormente, deve-se indicar ao **botão Guardar produto que deve invocar o método add\_produto() quando clicado**. Para isso retorna-se à definição do botão e adiciona o parâmetro *command*.

```
# Botão Adicionar Produto
self.botao_adicionar = ttk.Button(frame, text = "Guardar Produto", command =
self.add_produto)
```

3. Experimentar o que foi implementado (Executar a aplicação).



4. Agora serão implementadas **adicionar informação para serem armazenadas na base de dados**, e não simplesmente exibidas por consola.



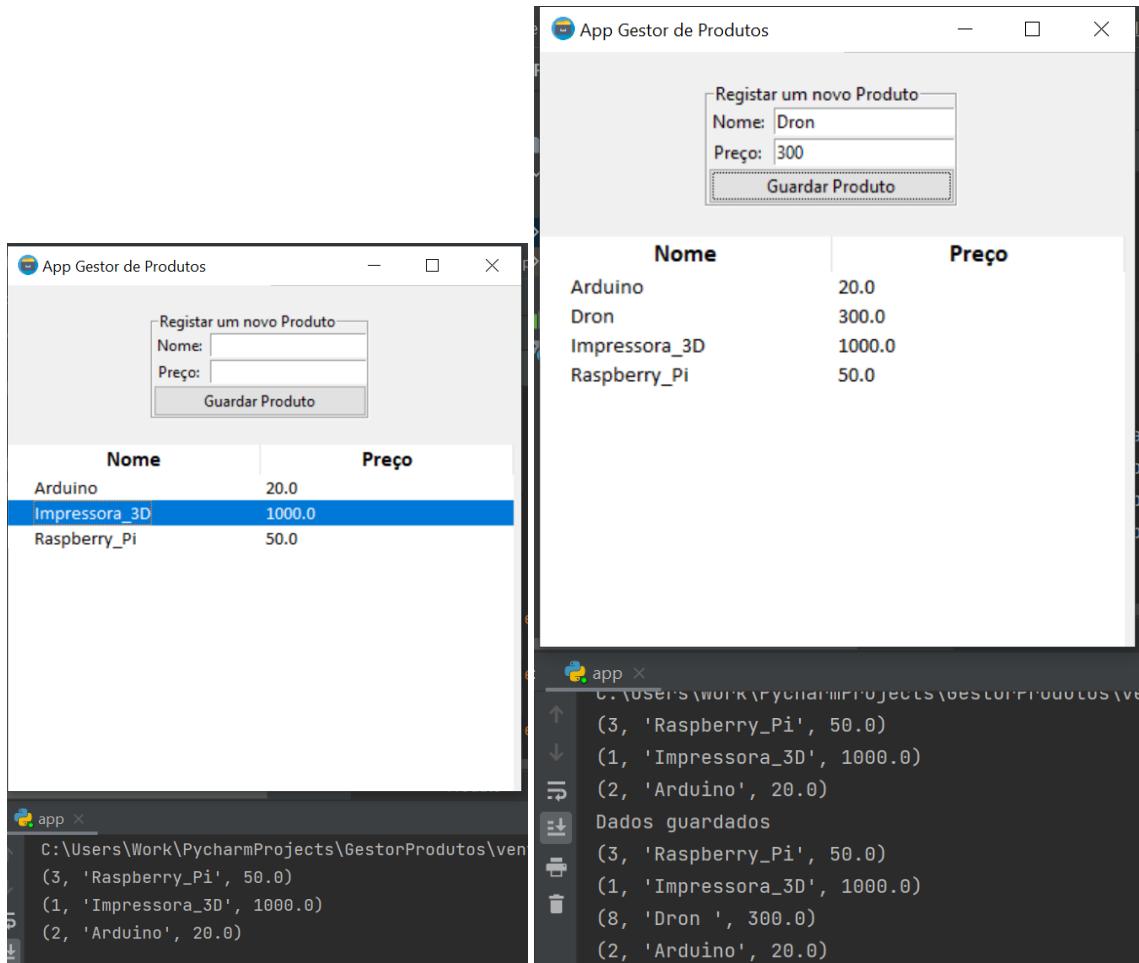
```
def add_produto(self):
    if self.validacao_nome() and self.validacao_preco():
        query = 'INSERT INTO produto VALUES(NULL, ?, ?)' # Consulta SQL (sem os
dados)
        parametros = (self.nome.get(), self.preco.get()) # Parâmetros da consulta
SQL
        self.db_consulta(query, parametros)
        print("Dados guardados")

        # Para debug
        #print(self.nome.get())
        #print(self.preco.get())
    elif self.validacao_nome() and self.validacao_preco() == False:
        print("O preço é obrigatório")
    elif self.validacao_nome() == False and self.validacao_preco():
        print("O nome é obrigatório")
    else:
        print("O nome e o preço são obrigatórios")

    self.get_produtos() # Quando se finalizar a inserção de dados voltamos a
invocar este método para atualizar o conteúdo e ver as alterações
```



## 5. Experimentar o que foi implementado (Executar a aplicação).



Como se pode ver, depois de inserir Dron (300) e clicar em **Guardar Produto**, foi adicionado à lista e pode-se comprovar visualmente, uma vez que a aplicação é atualizada após a inserção, ao chamar o método **get\_produtos()**. Mas deve-se verificar se está tudo correto na base de dados.



## 6. Comprovar na base de dados através do DB Browser

Antes:

The screenshot shows the DB Browser for SQLite interface. The main window displays a table named 'produto' with three rows:

id	nome	preco
1	Impressora_3D	1000.0
2	Arduino	20.0
3	Raspberry_Pi	50.0

The 'Raspberry\_Pi' row is currently selected. The right side of the interface shows the 'Edit Database Cell' panel where the value 'Raspberry\_Pi' is displayed. A tooltip indicates that the cell type is Text / Numeric and contains 12 character(s). Below this is the 'Remote' panel, which is currently empty.



Depois de clicar no botão de atualização

The screenshot shows the DB Browser for SQLite interface. The title bar reads "DB Browser for SQLite - C:\Users\Work\PycharmProjects\GestorProdutos\database\produtos.db". The main window displays a table named "produto" with the following data:

	id	nome	preco
1	1	Impressora_3D	1000.0
2	2	Arduino	20.0
3	3	Raspberry_Pi	50.0
4	8	Dron	300.0

An "Edit Database Cell" panel is open on the right, showing the value "1" in a cell. The "Mode" dropdown is set to "Text / Numeric". A tooltip indicates "Type of data currently in cell: Text / Numeric 1 character(s)". The "Remote" panel shows a connection to "DBHub.io". The bottom status bar shows "UTF-8 .db".

## 17. Melhorar add\_produto()

Neste ponto já se tem a funcionalidade de adicionar o produto perfeitamente implementada, mas há alguns detalhes que se podem melhorar:

- Fazer que,
- quando clicar no botão **Guardar produto**
  - O formulário é limpo automaticamente, para o deixar pronto para outra inserção.



- Que apareça uma mensagem de confirmação para o utilizador.
1. Volta-se ao construtor da classe **Produto**, e vai-se localizar uma mensagem informativa para o utilizador (**Label**) entre o botão **Guardar produto** e a tabela.

```
# Mensagem informativa para o utilizador
self.mensagem = Label(text = '', fg = 'red')
self.mensagem.grid(row = 3, column = 0, columnspan = 2, sticky = W + E)
```

Esta mensagem está inicialmente vazia, pelo que não se verá nada. Deve-se preencher com texto para se mostrar quando o utilizador clicar em **Guardar produto**.

2. Vai-se ao método **add\_produto()** e substitui o *print* ("Dados guardados") pela seguinte instrução, a qual irá preencher a Label com um texto a ser exibido.

```
self.mensagem['text'] = 'Produto {} adicionado com êxito'.format(self.nome.get())
# Label localizada entre o botão e a tabela
```

3. Para fazer que quando clicar em **Guardar produto** também **limpe o formulário**, será inserido logo abaixo da linha anterior o seguinte:

```
self.nome.delete(0, END) # Apagar o campo nome do formulário
self.preco.delete(0, END) # Apagar o campo preço do formulário
```

4. E serão modificadas as mensagens de validação (o preço é obrigatório, o nome é obrigatório, etc.)

```
self.mensagem['text'] = 'O preço é obrigatório'
...
self.mensagem['text'] = 'O nome é obrigatório'
...
self.mensagem['text'] = 'O nome e o preço são obrigatórios'
```



5. O método **add\_produto()** ficaria assim:

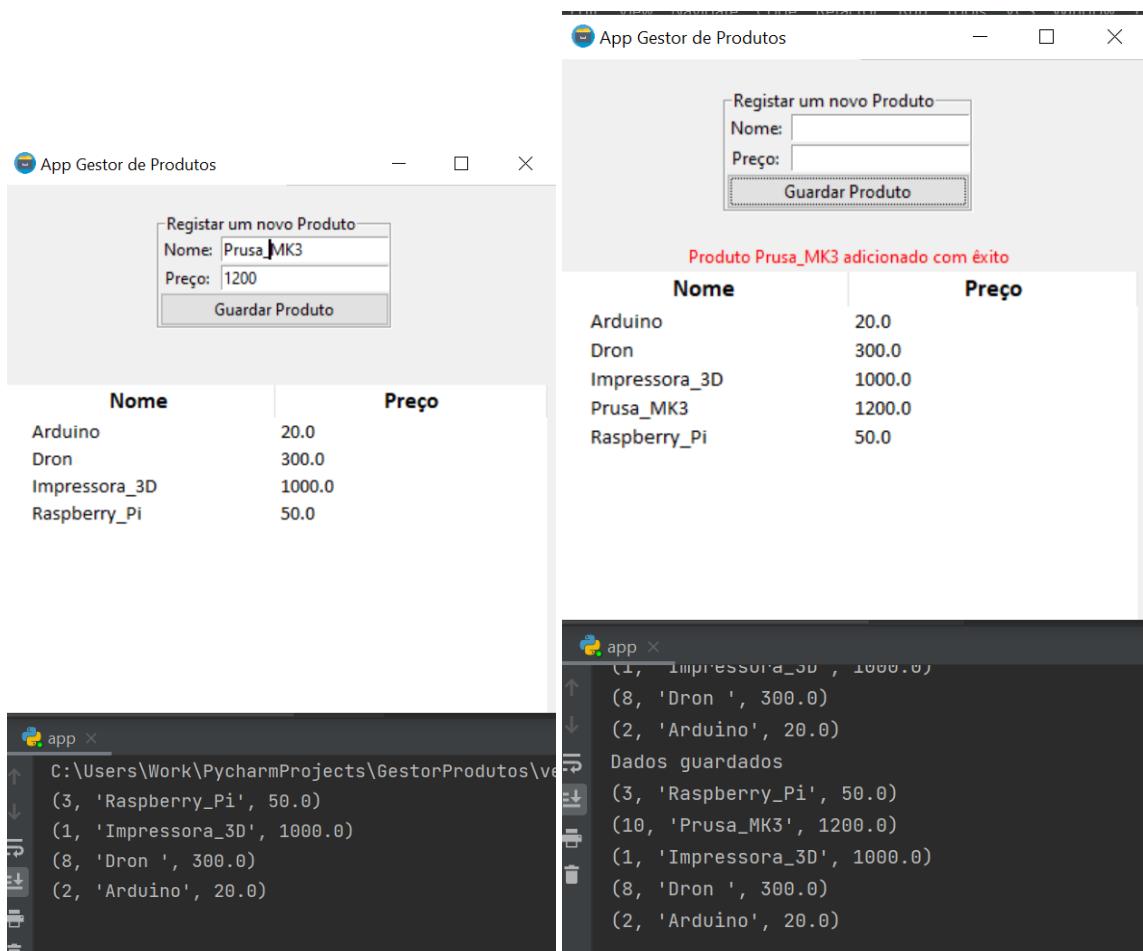
```
def add_produto(self):
    if self.validação_nome() and self.validação_preço():
        query = 'INSERT INTO produto VALUES(NULL, ?, ?)' # Consulta SQL (sem os
dados)
        parâmetros = (self.nome.get(), self.preço.get()) # Parâmetros da consulta
SQL
        self.db_consulta(query, parâmetros)
        print("Dados guardados")
        self.mensagem['text'] = 'Produto {} adicionado com
êxito'.format(self.nome.get()) # Label localizada entre o botão e a tabela
        self.nome.delete(0, END) # Apagar o campo nome do formulário
        self.preço.delete(0, END) # Apagar o campo preço do formulário

        # Para debug
        #print(self.nome.get())
        #print(self.preço.get())
    elif self.validação_nome() and self.validação_preço() == False:
        print("O preço é obrigatório")
        self.mensagem['text'] = 'O preço é obrigatório'
    elif self.validação_nome() == False and self.validação_preço():
        print("O nome é obrigatório")
        self.mensagem['text'] = 'O nome é obrigatório'
    else:
        print("O nome e o preço são obrigatórios")
        self.mensagem['text'] = 'O nome e o preço são obrigatórios'

    self.get_produtos() # Quando se finalizar a inserção de dados voltamos a
invocar a este método para atualizar o conteúdo e ver as alterações
```



6. Experimentar o que foi implementado (Executar a aplicação).





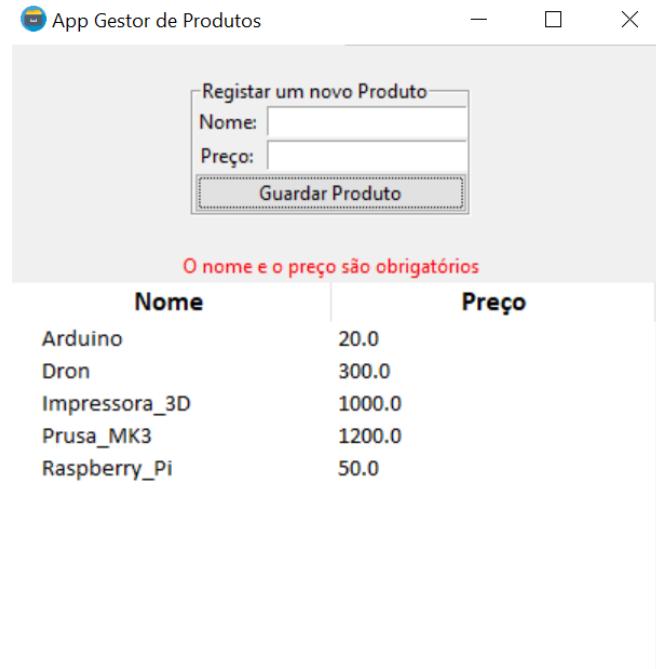
E testar as mensagens de validação:

**Left Screenshot (Price Validation Error):**

Nome	Preço
Arduino	20.0
Dron	300.0
Impressora_3D	1000.0
Prusa_MK3	1200.0
Raspberry_Pi	50.0

**Right Screenshot (Name Validation Error):**

Nome	Preço
Arduino	20.0
Dron	300.0
Impressora_3D	1000.0
Prusa_MK3	1200.0
Raspberry_Pi	50.0



## 18. Adicionar os dois botões que faltam: Eliminar e Editar

1. Deve-se ir ao construtor da classe **Produto**, e abaixo da definição da tabela, irão adicionar-se os dois botões

```
# Botões de Eliminar e Editar
botão_eliminar = ttk.Button(text = 'ELIMINAR')
botão_eliminar.grid(row = 5, column = 0, sticky = W + E)
botão_editar = ttk.Button(text='EDITAR')
botão_editar.grid(row = 5, column = 1, sticky = W + E)
```



2. Experimentar o que foi implementado (Executar a aplicação).



App Gestor de Produtos

— □ ×

Registrar um novo Produto

Nome:

Preço:

Guardar Produto

Nome	Preço
Arduino	20.0
Dron	300.0
Impressora_3D	1000.0
Prusa_MK3	1200.0
Raspberry_Pi	50.0

ELIMINAR EDITAR

## 19. Implementar a funcionalidade de Eliminar



1. Será criado um método chamado **del\_produto()** na classe Produtos. Deverá ser feito o seguinte:

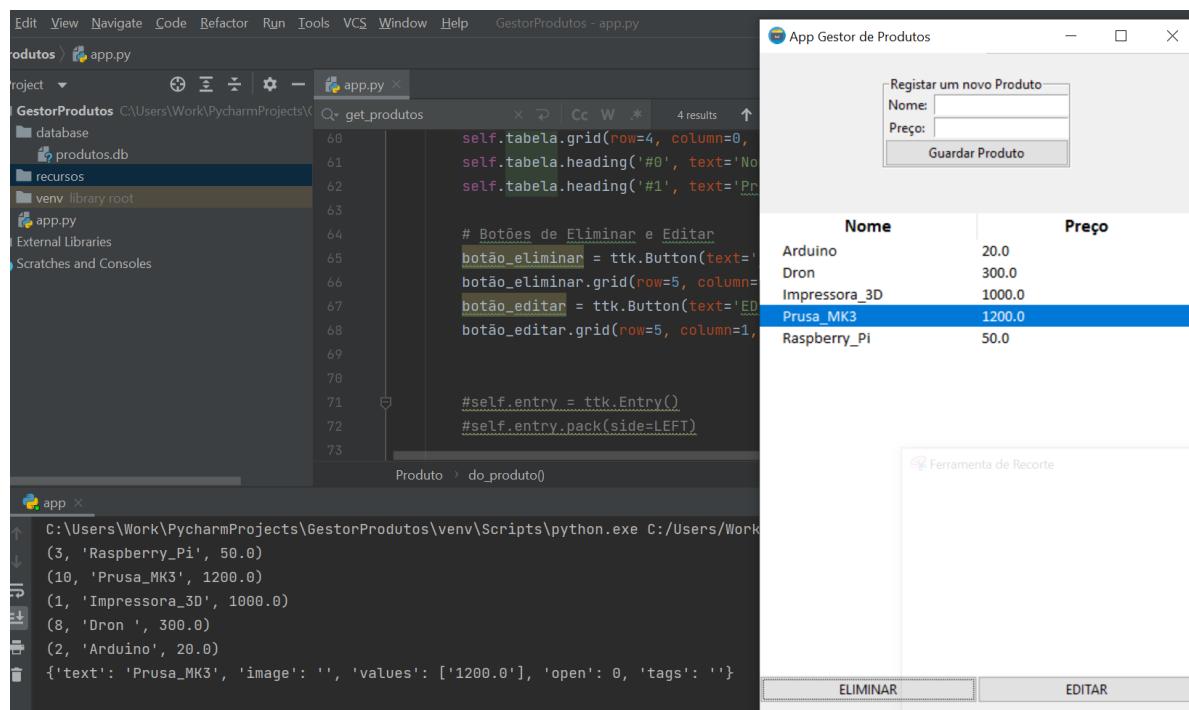
- O utilizador seleciona um produto da tabela
- O utilizador clica no botão ELIMINAR
- Mostram-se as informações do produto selecionado por consola

```
def del_produto(self):
    print(self.tabela.item(self.tabela.selection()))
```

2. O botão Eliminar é modificado, adicionando o atributo **command**

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

3. Experimentar o que foi implementado (Executar a aplicação). Verificar os dados que aparecem na consola depois de selecionar um produto e clicar em **Eliminar**

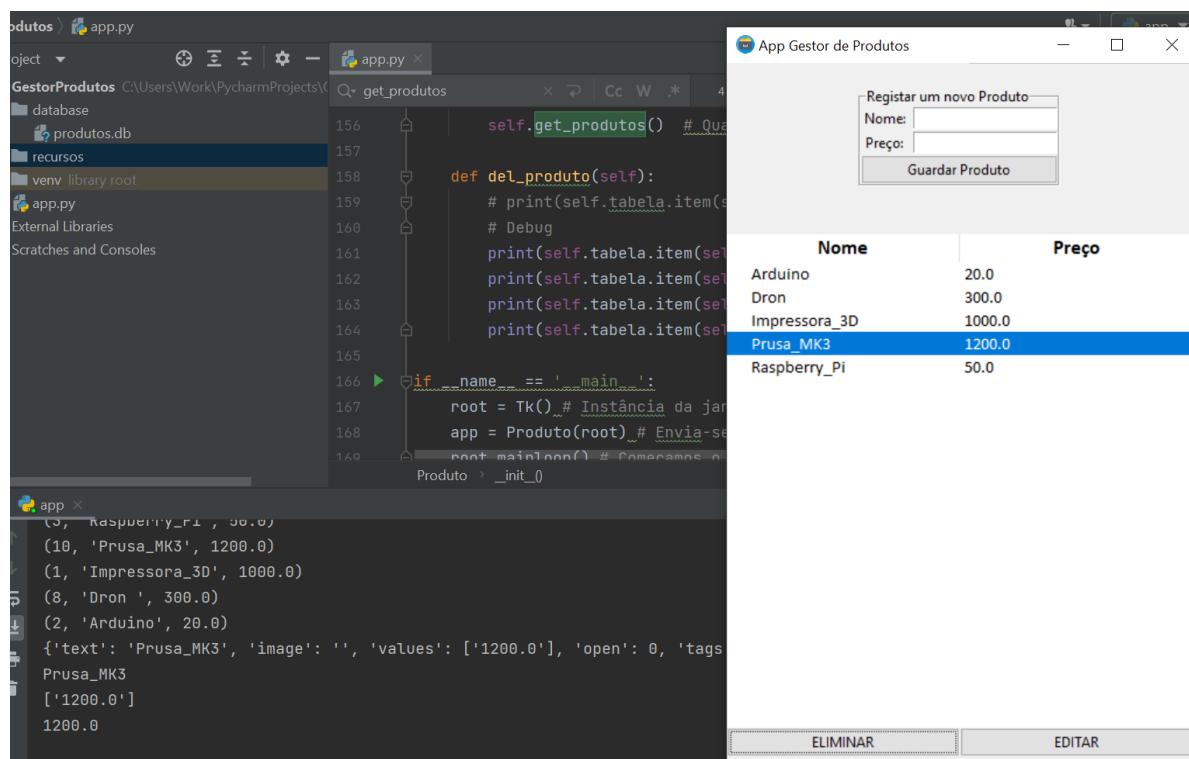




- Neste ponto, onde se está a extrair toda a informação do produto, vai-se aprofundar um pouco para ver como obter o valor do nome e preço (encontrado dentro de uma lista).

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

- Experimentar o que foi implementado (Executar a aplicação). Verificar os dados que aparecem na consola depois de selecionar um produto e clicar em **Eliminar**





6. Agora que o botão funciona e se tem acesso à informação, passa-se a fazer a implementação final. Serão comentadas as linhas anteriores para conservar para *debug*, e será adicionado o seguinte:

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

    self.mensagem['text'] = '' # Mensagem inicialmente vazio
    # Comprovação de que se selecione um produto para poder eliminá-lo
    try:
        self.tabela.item(self.tabela.selection())['text'][0]
    except IndexError as e:
        self.mensagem['text'] = 'Por favor, selecione um produto'
        return

    self.mensagem['text'] = ''
    nome = self.tabela.item(self.tabela.selection())['text']
    query = 'DELETE FROM produto WHERE nome = ?' # Consulta SQL
    self.db_consulta(query, (nome,))
    self.mensagem['text'] = 'Produto {} eliminado com êxito'.format(nome)
    self.get_produtos() # Atualizar a tabela de produtos
```

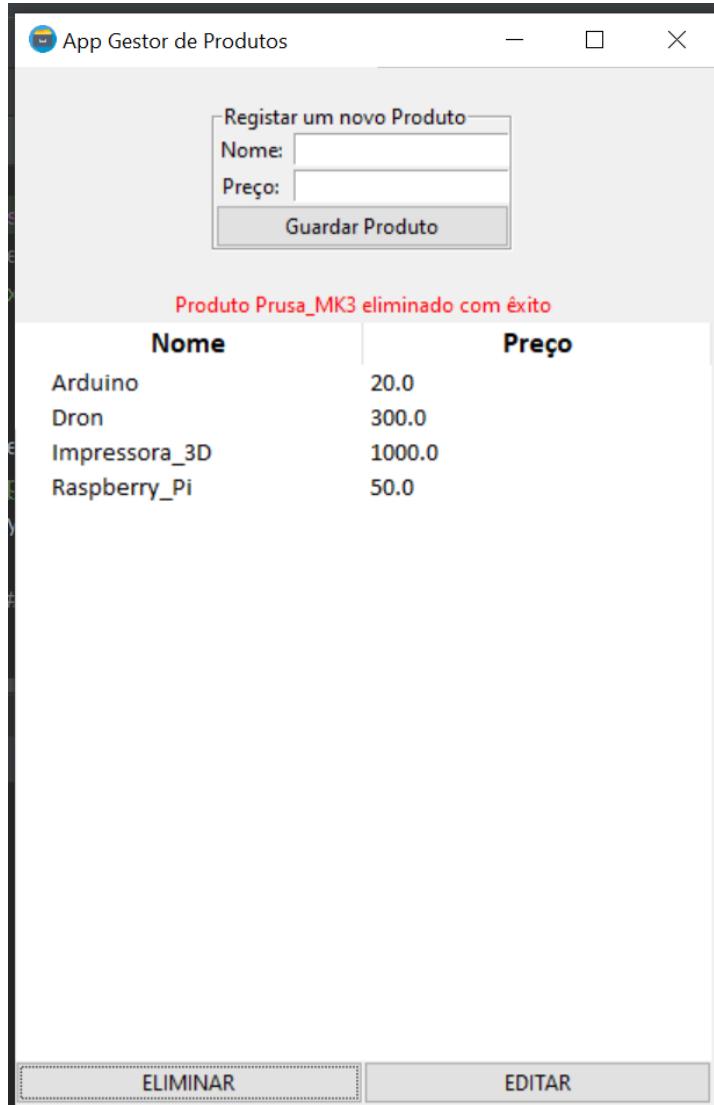


## 7. Experimentar o que foi implementado (Executar a aplicação).

The screenshot shows the PyCharm IDE interface with the following details:

- Project:** GestorProdutos
- app.py:** The code is shown in the editor, containing logic for deleting products from a database and displaying them in a table.
- Terminal:** Shows the command `python.exe` being run, followed by a list of tuples representing product data: `(3, 'Raspberry\_Pi', 50.0), (11, 'Prusa\_MK3', 1200.0), (1, 'Impressora\_3D', 1000.0), (8, 'Dron ', 300.0), (2, 'Arduino', 20.0)`.
- Running Application:** A window titled "App Gestor de Produtos" is displayed. It has a "Registrar um novo Produto" form with fields for Nome and Preço, and a "Guardar Produto" button. Below the form is a table showing product data:

Nome	Preço
Arduino	20.0
Dron	300.0
Impressora_3D	1000.0
Prusa_MK3	1200.0
Raspberry_Pi	50.0



Base de dados antes:



	<b>id</b>	<b>nome</b>	<b>preco</b>
	Filter	Filter	Filter
1	1	Impressora_3D	1000.0
2	2	Arduino	20.0
3	3	Raspberry_Pi	50.0
4	8	Dron	300.0
5	9	MK3	1200.0

Base de dados depois:

	<b>id</b>	<b>nome</b>	<b>preco</b>
	Filter	Filter	Filter
1	1	Impressora_3D	1000.0
2	2	Arduino	20.0
3	3	Raspberry_Pi	50.0
4	8	Dron	300.0

## 20. Implementar a funcionalidade de Editar



1. Será criado um método chamado **edit\_produto()** dentro da classe Produto. Será muito semelhante a eliminar produto, com a diferença de que vamos criar uma segunda janela para poder modificar os dados de um produto:

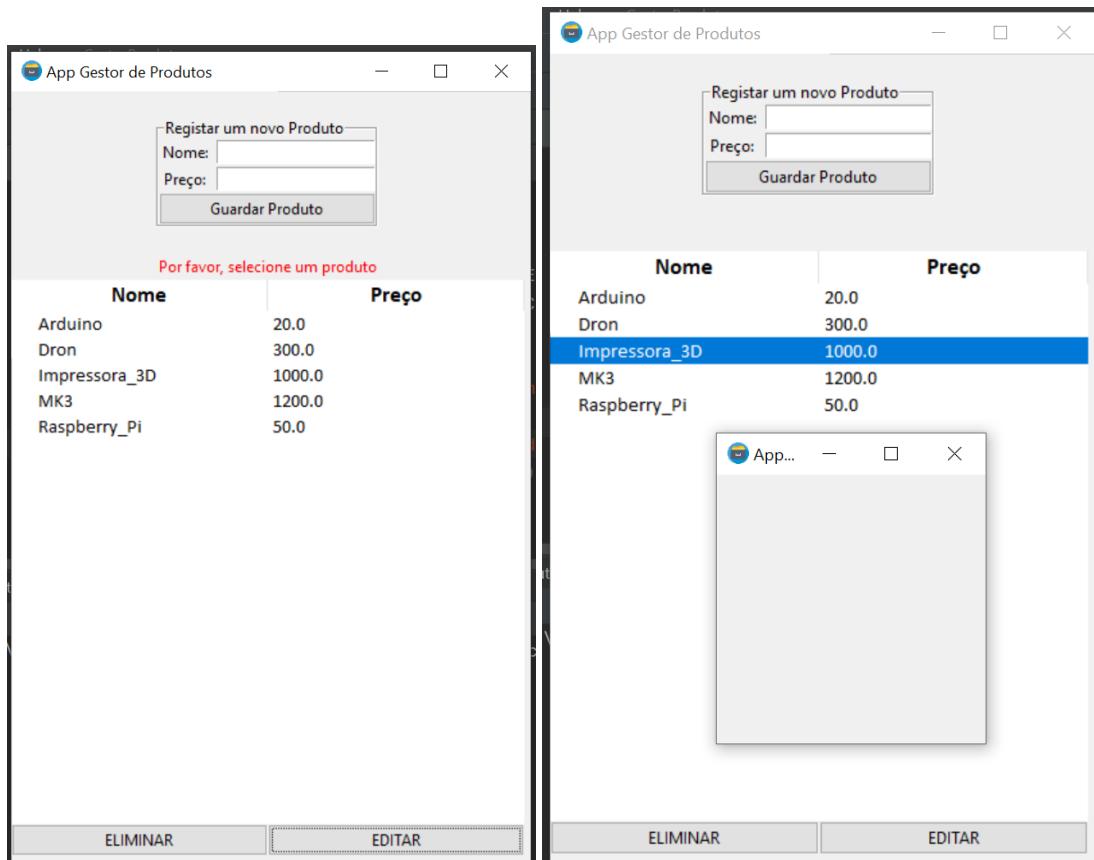
```
def edit_produto(self):
    self.mensagem['text'] = '' # Mensagem inicialmente vazia
    try:
        nome = self.tabela.item(self.tabela.selection())['text'][0]
    except IndexError as e:
        self.mensagem['text'] = 'Por favor, selecione um produto'
        return
    old_preco = self.tabela.item(self.tabela.selection())['values'][0] # O preço
encontra-se dentro de uma lista

    self.janela_editar = Toplevel() # Criar uma janela à frente da principal
    self.janela_editar.title = "Editar Produto" # Titulo da janela
    self.janela_editar.resizable(1, 1) # Ativar a redimensão da janela. Para
desativá-la: (0,0)
    self.janela_editar.wm_iconbitmap('recursos/icon.ico') # Ícone da janela
```

2. O botão **Editar** é modificado, adicionando o atributo **command**

```
botão_editar = ttk.Button(text='EDITAR', command = self.edit_produto)
```

3. Experimentar o que foi implementado (Executar a aplicação).





4. Logo após a criação da segunda janela, será adicionado o seguinte código que cobre os seguintes aspectos:

- Um título para esta nova janela
- Um *frame* que englobe:
  - O nome antigo (sem que se possa modificar)
  - O preço antigo (sem que se possa modificar)
  - O nome novo
  - O preço novo
  - Um botão para atualizar o produto

O comportamento ideal seria:

- Se o utilizador modifica o nome e o preço
  - Modifica-se o nome e preço (tanto na tabela como na base de dados)
- Se o utilizador modifica apenas o nome
  - Modifica-se apenas o nome, conservando o preço antigo
- Se o utilizador modifica apenas o preço
  - Modifica-se apenas o preço, conservando o nome antigo
- Se o utilizador não modifica nada
  - Não se modifica nada, conserva-se o nome e preço antigo

Em todos os casos, o utilizador deve ser notificado das alterações com um texto no ecrã.

```
# Janela nova (editar produto)
self.janela_editar = Toplevel() # Criar uma janela à frente da principal
self.janela_editar.title = "Editar Produto" # Título da janela
self.janela_editar.resizable(1, 1) # Ativar o redimensionamento da
janela. Para desativá-la: (0,0)
self.janela_editar.wm_iconbitmap('recursos/icon.ico') # Ícone da janela

título = Label(self.janela_editar, text='Edição de Produtos',
font=('Calibri', 50, 'bold'))
título.grid(column=0, row=0)

# Criação do recipiente Frame da janela de Editar Produto
frame_ep = LabelFrame(self.janela_editar, text="Editar o seguinte
Produto") # frame_ep: Frame Editar Produto
frame_ep.grid(row=1, column=0, columnspan=20, pady=20)
```



```
# Label Nome antigo
self.etiqueta_nome_antigo = Label(frame_ep, text="Nome antigo: ") #
Etiqueta de texto localizada no frame
self.etiqueta_nome_antigo.grid(row=2, column=0) # Posicionamento através
de grid
# Entry Nome antigo (texto que não se poderá modificar)
self.input_nome_antigo = Entry(frame_ep,
textvariable=StringVar(self.janela_editar, value=nome),
state='readonly')
self.input_nome_antigo.grid(row=2, column=1)

# Label Nome novo
self.etiqueta_nome_novo = Label(frame_ep, text="Nome novo: ")
self.etiqueta_nome_novo.grid(row=3, column=0)
# Entry Nome novo (texto que se poderá modificar)
self.input_nome_novo = Entry(frame_ep)
self.input_nome_novo.grid(row=3, column=1)
self.input_nome_novo.focus() # Para que a seta do rato vá a esta Entry ao
início

# Label Preço antigo
self.etiqueta_preco_antigo = Label(frame_ep, text="Preço antigo: ") #
Etiqueta de texto localizada no frame
self.etiqueta_preco_antigo.grid(row=4, column=0) # Posicionamento através
de grid
# Entry Preço antigo (texto que não se poderá modificar)
self.input_preco_antigo = Entry(frame_ep,
textvariable=StringVar(self.janela_editar, value=old_preco),
state='readonly')
self.input_preco_antigo.grid(row=4, column=1)

# Label Preço novo
self.etiqueta_preco_novo = Label(frame_ep, text="Preço novo: ")
self.etiqueta_preco_novo.grid(row=5, column=0)
# Entry Preço novo (texto que se poderá modificar)
self.input_preco_novo = Entry(frame_ep)
self.input_preco_novo.grid(row=5, column=1)

# Botão Atualizar Produto
self.botão_atualizar = ttk.Button(frame_ep, text="Atualizar Produto",
command=lambda:
self.atualizar_produtos(self.input_nome_novo.get(),
self.input_nome_antigo.get(),
self.input_preco_novo.get(),
```



```
self.input_preco_antigo.get()))

self.botao_atualizar.grid(row=6, columnspan=2, sticky=W + E)
self.etiqueta_preco_novo.grid(row=5, column=0)
# Entry Preço novo (texto que se poderá modificar)
self.input_preco_novo = Entry(frame_ep)
self.input_preco_novo.grid(row=5, column=1)

# Botão Atualizar Produto
self.botão_atualizar = ttk.Button(frame_ep, text="Atualizar Produto",
                                  command=lambda:
self.atualizar_produtos(self.input_nome_novo.get(),
self.input_nome_antigo.get(),
self.input_preco_novo.get(),
self.input_preco_antigo.get()))

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

5. E, finalmente, implementa-se o método **atualizar\_produtos()** que é chamado de **edit\_produto()**, e que tem como objetivo:

- Executar a consulta SQL
- Modificar o texto que aparece no ecrã para retroalimentar o utilizador
- Fechar a janela para editar produtos
- Atualizar a tabela de produtos da aplicação (consultando novamente a base de dados)

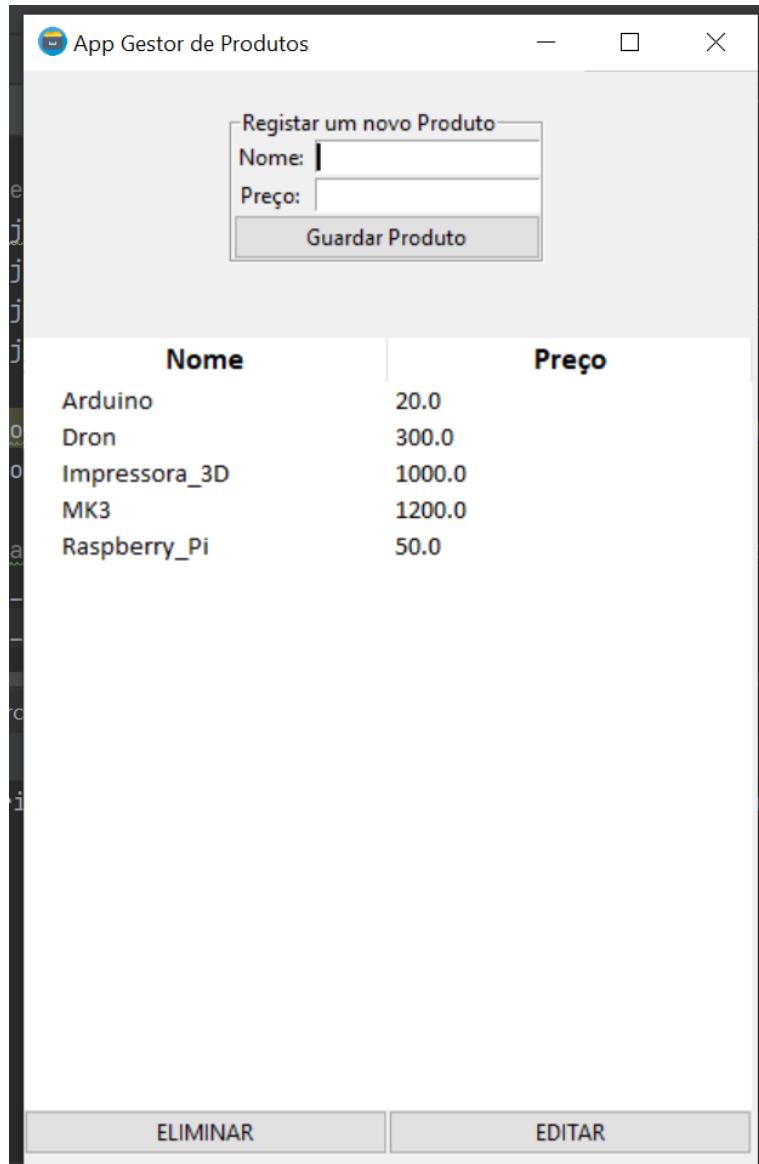
```
def atualizar_produtos(self, novo_nome, antigo_nome, novo_preco,
antigo_preco):
    produto_modificado = False
    query = 'UPDATE produto SET nome = ?, preco = ? WHERE nome = ? AND
preco = ?'
    if novo_nome != '' and novo_preco != '':
        # Se o utilizador escreve novo nome e novo preço, mudam-se ambos
        parametros = (novo_nome, novo_preco, antigo_nome, antigo_preco)
        produto_modificado = True
    elif novo_nome != '' and novo_preco == '':
        # Se o utilizador deixa vazio o novo preço, mantém-se o preço
        anterior
        parametros = (novo_nome, antigo_preco, antigo_nome, antigo_preco)
        produto_modificado = True
    elif novo_nome == '' and novo_preco != '':
        # Se o utilizador deixa vazio o novo nome, mantém-se o nome
        anterior
```

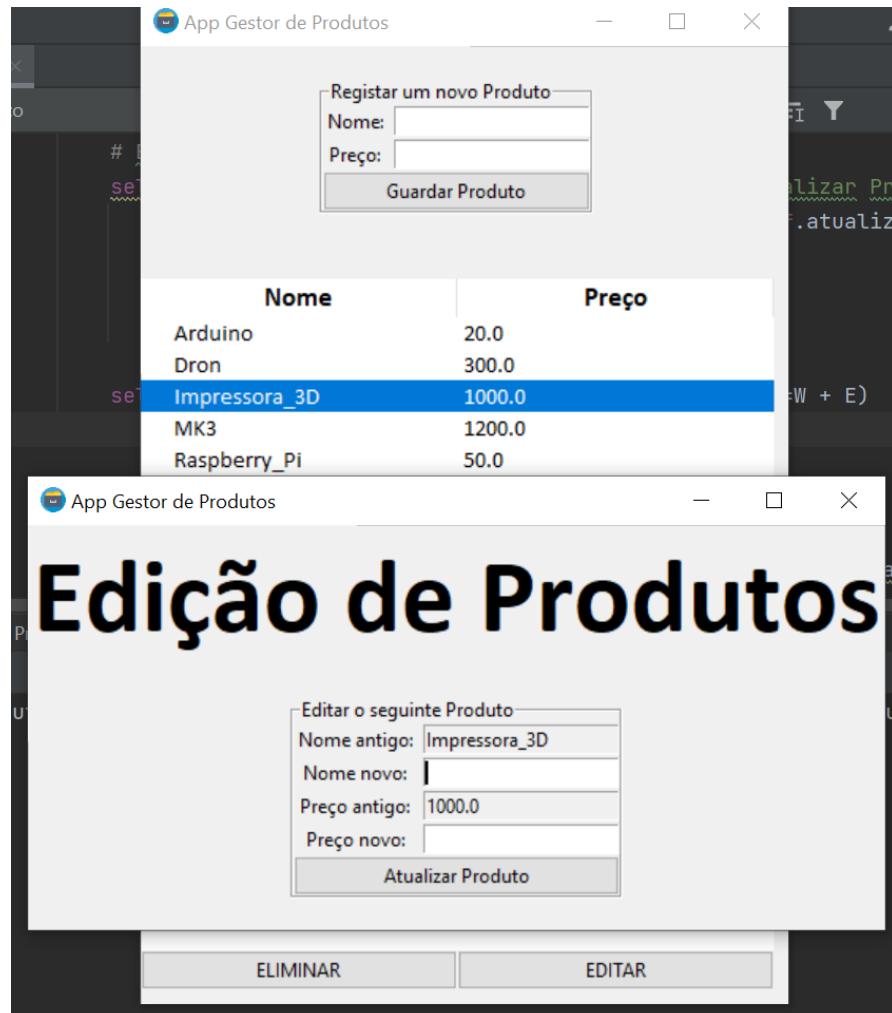


```
parametros = (antigo_nome, novo_preco, antigo_nome, antigo_preco)
produto_modificado = True

if (produto_modificado):
    self.db_consulta(query, parametros) # Executar a consulta
    self.janela_editar.destroy() # Fechar a janela de edição de
produtos
    self.mensagem['text'] = 'O produto {} foi atualizado com
êxito'.format(
        antigo_nome) # Mostrar mensagem para o utilizador
    self.get_produtos() # Atualizar a tabela de produtos
else:
    self.janela_editar.destroy() # Fechar a janela de edição de
produtos
    self.mensagem['text'] = 'O produto {} NÃO foi atualizado'.format(
        antigo_nome) # Mostrar mensagem para o utilizador
```

6. Experimentar o que foi implementado (Executar a aplicação).







Modifica-se o produto e clica-se em Atualizar Produto

App Gestor de Produtos

# Edição de Produtos

Editar o seguinte Produto

Nome antigo:	Impressora_3D
Nome novo:	
Preço antigo:	1000.0
Preço novo:	

Atualizar Produto

Resultado final na aplicação:



Resultado final na base de dados (se não se veem as alterações a priori, atualizar o conteúdo):



	id	nome	preco
1	1	Impressora_3D	999.0
2	2	Arduino	20.0
3	3	Raspberry_Pi	50.0
4	8	Dron	300.0

Possíveis evoluções desta aplicação:

- Implementar o SQLAlchemy sobre SQLite para que possa trocar de bases de dados sem tocar no código.
- Adicionar mais uma secção aos **Produtos**, que seja a Categoria. Para fazer isso, terá de modificar o formulário de inserção de produtos, a listagem de produtos, o modelo de base de dados, etc.
- Adicionar algum novo widget gráfico proporcionado por Tkinter, que não seja uma *label*, botão, caixa de texto ou uma tabela.



## 21. Melhorar o desenho

- Irá aplicar-se estilos aos diferentes textos que tem na aplicação. Adicionar o seguinte estilo:

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

Para os textos simples mais importantes:

```
frame = LabelFrame(self.janela, text = "Registar um novo Produto",
font=('Calibri', 16, 'bold'))

frame_ep = LabelFrame(self.janela_editar, text="Editar o seguinte Produto",
font=('Calibri', 16, 'bold'))
```

E o próximo estilo para o resto dos textos (nome, preço, etc.)

```
font=('Calibri', 13)

self.etiqueta_nome = Label(frame, text="Nome: ", font=('Calibri', 13))
self.nome = Entry(frame, font=('Calibri', 13))

self.etiqueta_preco = Label(frame, text="Preço: ", font=('Calibri', 13))
self.preco = Entry(frame, font=('Calibri', 13))

self.etiqueta_nome_antigo = Label(frame_ep, text="Nome antigo: ",
font=('Calibri', 13))

self.input_nome_antigo = Entry(frame_ep,
textvariable=StringVar(self.janela_editar, value=nome), state='readonly',
font=('Calibri', 13))

self.etiqueta_nome_novo = Label(frame_ep, text="Nome novo: ", font=('Calibri', 13))
self.input_nome_novo = Entry(frame_ep, font=('Calibri', 13))

self.etiqueta_preco_antigo = Label(frame_ep, text="Preço antigo: ",
font=('Calibri', 13))
```



```
self.input_preco_antigo = Entry(frame_ep,
textvariable=StringVar(self.janela_editar, value=old_preco), state='readonly',
font=('Calibri', 13))

self.etiqueta_preco_novo = Label(frame_ep, text="Preço novo: ", font=('Calibri', 13))

self.input_preco_novo = Entry(frame_ep, font=('Calibri', 13))
```

2. Modificar os botões da seguinte maneira:

```
# Botão Adicionar Produto
s = ttk.Style()
s.configure('my.TButton', font=('Calibri', 14, 'bold'))
self.botao_adicionar = ttk.Button(frame, text="Guardar Produto",
command=self.add_produto, style='my.TButton')
self.botao_adicionar.grid(row=3, columnspan=2, sticky=W + E)
```

```
# Botões de Eliminar e Editar
s = ttk.Style()
s.configure('my.TButton', font=('Calibri', 14, 'bold'))
botao_eliminar = ttk.Button(text = 'ELIMINAR', command = self.del_produto,
style='my.TButton')
botao_eliminar.grid(row = 5, column = 0, sticky = W + E)
botao_editar = ttk.Button(text='EDITAR', command = self.edit_produto,
style='my.TButton')
botao_editar.grid(row = 5, column = 1, sticky = W + E)
```

```
# Botão Atualizar Produto
s = ttk.Style()
s.configure('my.TButton', font=('Calibri', 14, 'bold'))
self.botao_atualizar = ttk.Button(frame_ep, text="Atualizar Produto",
style='my.TButton',
command=lambda:
self.atualizar_produtos(self.input_nome_novo.get(),
self.input_nome_antigo.get(),
self.input_preco_novo.get(),
self.input_preco_antigo.get()))
self.botao_atualizar.grid(row=6, columnspan=2, sticky=W + E)
```



## 22. Resultado final

App Gestor de Produtos

### Registrar um novo Produto

Nome:

Preço:

**Guardar Produto**

Nome	Preço
Arduino	20.0
Dron	300.0
Impressora_3D	999.0
Raspberry_Pi	50.0

**ELIMINAR** **EDITAR**





Ou se retirasse o título da Edição do Produtos, o resultado final seria:

The screenshot shows a Windows application window titled "App Gestor de Produtos". Inside, there are two forms:

- Registrar um novo Produto:** A form with fields for "Nome" and "Preço", and a "Guardar Produto" button.
- Lista de Produtos:** A table with columns "Nome" and "Preço", containing the following data:

Nome	Preço
Arduino	20.0
Dron	300.0
Impressora_3D	999.0
Raspberry_Pi	50.0
- Editar o seguinte Produto:** A form with fields for "Nome antigo" (set to "Dron"), "Nome novo" (empty), "Preço antigo" (set to "300.0"), and "Preço novo" (empty), and an "Atualizar Produto" button.
- Action Buttons:** At the bottom are two buttons: "ELIMINAR" and "EDITAR".



## 23. Bibliografia

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 (documentação oficial)

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

DB Browser for SQLite (interface gráfica para SQLite)

<https://sqlitebrowser.org/>

Manual (não oficial) de Tkinter

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

Manual (não oficial) de Tkinter. Secção widgets

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