EGOTECHWORLD
egotechworld.com

Simple POS System

Build a Real Point-of-Sale App with Python

PYTHON TKINTER SQLITE

Beginner-Friendly Tutorial

Step-by-step build of a working Point-of-Sale system
Includes installation, code walkthrough, ER diagram,
software testing, packaging as a setup file, and
practical tips on selling and deploying to a real client.

Welcome

Welcome to this beginner-friendly tutorial from **EGOTECHWORLD PVT LTD** (egotechworld.com).

In this guide you will build a complete **Simple Point-of-Sale (POS) system** using Python, Tkinter, and SQLite — three tools that come either built into Python or are tiny to install. No web hosting, no complicated setup, no paid software needed.

By the end you will have a real desktop application that a small shop can actually use to sell items, print bills, manage products, and view daily sales reports. You will also learn how to package it into a Windows setup file (.exe installer) and hand it over to a paying client.

Who is this for?

- Students learning Python who want a real, finished project for their portfolio.
- Teachers and trainers who need a complete project example for class.
- Junior developers who want to sell their first software product.
- Shop owners curious about how POS software actually works inside.

What you will learn

- What a POS system is and why every shop needs one.
- How to design a database (ER diagram) before writing code.
- Python + Tkinter desktop UI step by step.
- SQLite — a single-file database that needs no server.
- Software testing basics with Python's unittest.
- Packaging your app into a Windows installer with PyInstaller and Inno Setup.
- Pricing, selling, and deploying the software to a real customer.

> ***NOTE:*** *You only need a basic understanding of Python (variables, functions, if/else, lists). If you can write a "Hello World" program, you can finish this tutorial.*

Chapter 1 — What is a POS System?

POS stands for **Point of Sale**. It is the place — and the software — where a sale happens. When you go to a supermarket, the cashier scans your items, the screen shows prices, the total appears, you pay, and a printed bill comes out.

That whole flow is run by a POS system. A POS system is the modern replacement for an old cash register and a handwritten bill book.

Instead of paper, every sale is saved into a database, which means the shop owner can later answer questions like:

- "How much did I sell today, this week, this month?"
- "Which products are selling fast and which are sitting unsold?"
- "How much stock of sugar do I have left right now?"
- "Which cashier handled the most sales yesterday?"

1.1 Why every shop needs a POS

Even small shops in Sri Lanka and around the world are moving away from paper bill books because a POS system gives real, measurable benefits. The table below compares life with and without a POS system.

Without POS	With POS
Hand-written bills, slow	One click bill, fast
Stock counted manually monthly	Stock auto-updated on every sale
No idea about daily profit	Sales report at end of each day
Easy for cashiers to make mistakes	Calculator does maths automatically
Cannot answer "best-selling item?"	Reports show top items instantly
Lost bills = lost records	Database keeps every sale forever

1.2 Types of POS systems

There are three common types you will see in real businesses:

- **Desktop POS** — one computer in the shop runs the software. Works offline. This is what we are building.
- **Cloud POS** — software runs on a website. Multiple branches can connect. Needs internet all the time.
- **Mobile POS** — phone or tablet acts as the cash register. Common for restaurants and food trucks.

For a single small shop, a **desktop POS** is the best choice — it is simple, runs offline, and never stops working when the internet is down.

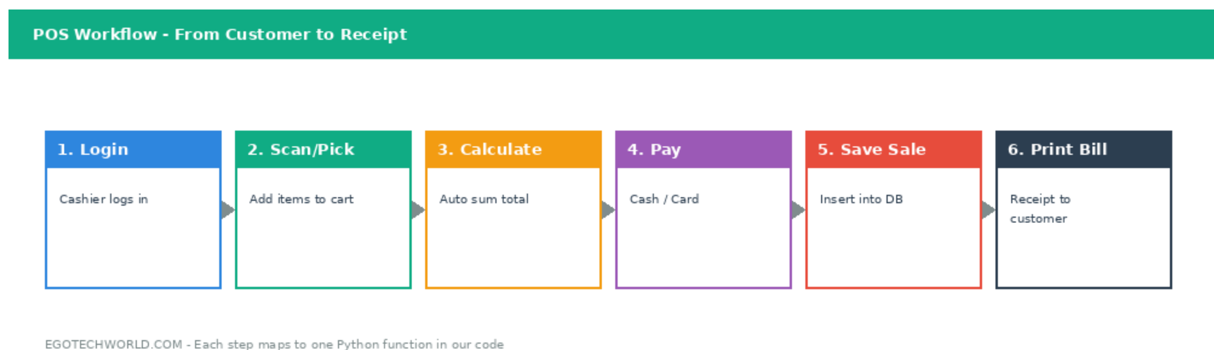

Figure 1.1 — The 6-step POS workflow

1.3 What our Simple POS will do

We will build a system with the features below.

- **Login screen** — only authorized users can open the POS.
- **Product management** — add, edit, delete products with price and stock.
- **Sales screen** — pick items, build a cart, see live total.
- **Bill printing** — generate a clean text receipt.
- **Sales history** — see every sale ever made.
- **Daily reports** — total sales for any selected day.
- **SQLite database** — all data saved in a single *pos.db* file.

Preview of the Final App

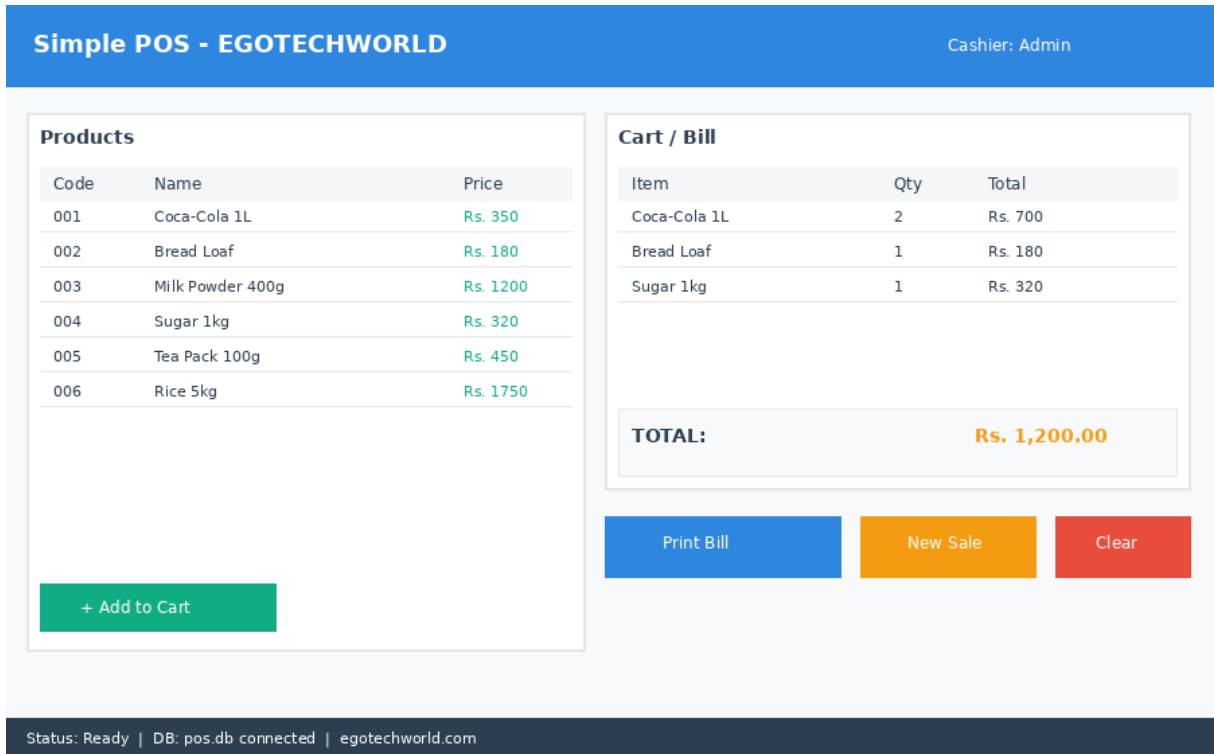

Figure 1.2 — The finished POS app you will build by the end of this tutorial

Chapter 2 — Development Tools You Need

Before writing any code, set up the small set of free tools below. Everything in this list is free and works on Windows, macOS, and Linux.

Tool	Purpose	Where to get it
Python 3.11+	The programming language	python.org/downloads
VS Code	Code editor	code.visualstudio.com
DB Browser for SQLite	View the database visually	sqlitebrowser.org
Git (optional)	Save versions of your code	git-scm.com
PyInstaller	Convert .py to .exe	pip install pyinstaller
Inno Setup (Windows)	Build a real .exe installer	jrsoftware.org/isdl.php

2.1 Installing Python (step by step)

On Windows:

- Go to **python.org/downloads** and click "Download Python 3.x".
- Run the installer.
- **VERY IMPORTANT:** tick the box that says *"Add python.exe to PATH"* at the bottom of the first screen.
- Click *Install Now* and wait until it finishes.

Now open **Command Prompt** (press Windows + R, type *cmd*, press Enter) and verify Python is installed:

```
python --version
pip --version
```

You should see something like *Python 3.12.x*. If you see "command not found", you forgot to tick the PATH box — uninstall and reinstall, this time ticking the box.

2.2 Installing VS Code and the Python extension

VS Code is the editor we will use to write our code. Download it from **code.visualstudio.com** and install it.

After installation, set up the helpful extensions:

- Open VS Code and click the Extensions icon on the left side (4 squares).
- Search for **"Python"** by Microsoft and click *Install*.
- Search for **"SQLite Viewer"** and install it too — this lets you peek inside *.db* files without leaving the editor.

2.3 Built-in tools (no install needed)

Two of our three main technologies need no installation at all because they ship with Python:

- **tkinter** — Python's built-in GUI library. Comes with Python on Windows and macOS. On Linux you may need *sudo apt install python3-tk*.
- **sqlite3** — Python's built-in database driver. Always available, no separate database server needed.

Test that both work by running these commands:

```
python -c "import tkinter; tkinter._test()"
python -c "import sqlite3; print(sqlite3.sqlite_version)"
```

The first command opens a tiny test window — close it. The second command prints the SQLite version (something like 3.45.x). If both work, you are ready to build.

Chapter 3 — Database Plan & ER Diagram

Before writing a single line of Python, we plan the database. This is the most important step in any business application.

A bad database is extremely painful to fix later; a good database makes the rest of the code almost write itself. So spend time on this chapter — it pays off in every chapter that follows.

3.1 What information does a POS need to store?

Think about a real shop. Every sale involves four kinds of "things":

- **Products** being sold (the items on shelves).
- **Users** doing the selling (cashiers and admins).
- **Sales** — one row per bill (the receipt header).
- **Sale items** — the individual lines inside that bill.

Each of these "things" becomes a **table** in our database.

Table 1: products

Stores everything the shop sells.

```
id          INTEGER  PRIMARY KEY AUTOINCREMENT
code        TEXT     UNIQUE       (like "P001", or barcode)
name        TEXT     NOT NULL     (e.g. "Coca-Cola 1L")
price       REAL     NOT NULL     (selling price)
stock       INTEGER  DEFAULT 0    (how many left)
category    TEXT                  (e.g. "Drinks", "Bakery")
```

Table 2: users

Stores cashiers and admin accounts.

```
id          INTEGER  PRIMARY KEY AUTOINCREMENT
username    TEXT     UNIQUE   NOT NULL
password    TEXT     NOT NULL    (hashed)
role        TEXT                 ("admin" or "cashier")
full_name   TEXT
```

Table 3: sales

One row per bill — the "header" of a sale.

```
id           INTEGER  PRIMARY KEY AUTOINCREMENT
bill_no      TEXT     UNIQUE   (e.g. "INV-20260507-001")
total        REAL     NOT NULL
cashier_id   INTEGER  FK -> users.id
sale_date    DATETIME DEFAULT CURRENT_TIMESTAMP
```

Table 4: sale_items

The actual lines on the bill — one row per product per sale.

```
id           INTEGER  PRIMARY KEY AUTOINCREMENT
sale_id      INTEGER  FK -> sales.id
product_id   INTEGER  FK -> products.id
quantity     INTEGER  NOT NULL
unit_price   REAL     NOT NULL
subtotal     REAL     NOT NULL
```

3.2 ER Diagram (Entity Relationship)

An ER diagram shows tables and the relationships (lines) between them. Reading the diagram below:

- One **sale** can contain many **sale_items** — the famous "one-to-many" relationship.
- One **product** can appear in many **sale_items**.
- One **user** (cashier) can create many **sales**.

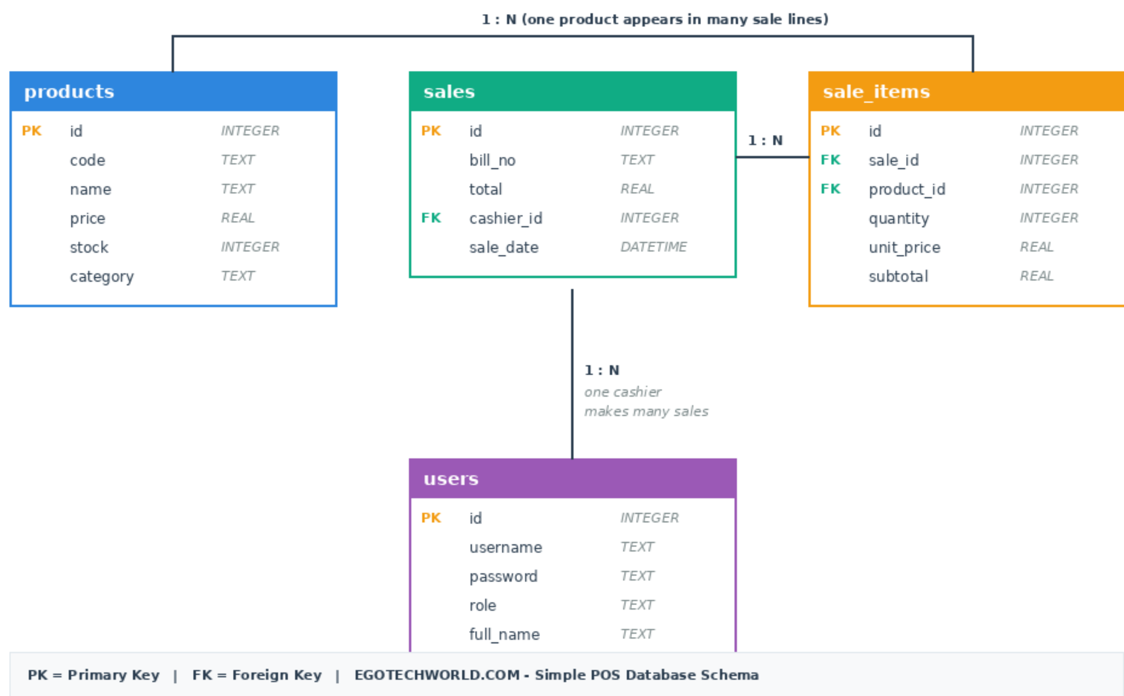

Figure 3.1 — ER Diagram of our POS database (4 tables)

3.3 Why split into 4 tables?

A common beginner mistake is to put everything into one giant table — product name, price, quantity, customer, date — all in one row.

This is called **denormalization** and it causes huge problems:

- If you change a product's name, you have to update every old sale.
- You waste storage repeating the product name on every line.
- Reports become very slow.
- It is impossible to track which products were never sold.

Splitting into related tables (**normalization**) is the proper way. It is the same approach used by every commercial POS in the world.

Chapter 4 — Project Folder Structure

Good software is organized into folders and files that each have one clear purpose. Below is the structure we will use.

Create this exact folder layout on your computer before starting Chapter 5. Doing this upfront keeps your project clean as it grows.

Project Folder Structure

```
simple_pos/                          # main project folder
├── main.py                          # app entry point
├── database.py                      # all DB functions
├── ui/                              # window screens folder
│   ├── login_window.py              # login UI
│   ├── pos_window.py                # main POS UI
│   ├── product_window.py            # manage products
│   └── reports_window.py            # sales reports
├── data/
│   └── pos.db                       # SQLite database file
├── assets/
│   └── logo.png                     # company logo
├── tests/
│   └── test_database.py             # unit tests
├── seed.py                          # sample data loader
├── requirements.txt                 # pip packages list
├── README.md                        # docs for client
└── setup_installer.iss              # Inno Setup script
```

Figure 4.1 — Final folder structure of our project

4.1 Why this structure?

- **main.py** is the front door — running this file starts the whole app.
- **database.py** contains every SQL operation. Keeping all SQL in one place means if you change the database later, you only edit one file.
- **ui/** folder holds all the windows. Each screen of the app gets its own file.
- **data/pos.db** is the actual database file. It is created automatically on first run.
- **tests/** contains automated tests so you can verify nothing broke after a change.
- **requirements.txt** is a list of pip packages — important so a different computer can install the same versions.
- **setup_installer.iss** is the script we will use later to build a professional Windows installer.

4.2 Create the folders now

Open a terminal (Command Prompt on Windows) in the place where you want the project, and run:

```
mkdir simple_pos
cd simple_pos
mkdir ui data assets tests
echo. > main.py
echo. > database.py
echo. > requirements.txt
```

NOTE: *On macOS or Linux, replace echo. > with touch (for example touch main.py).*

Chapter 5 — Building database.py

We start by writing the database module. This file does three jobs:

- Creates the *.db* file and all tables on first run.
- Provides simple Python functions for adding and reading data.
- Handles all the SQL so the rest of the code never touches SQL directly.

5.1 Creating the connection helper

Open **database.py** and paste the code below:

```
import sqlite3
import os
from datetime import datetime

DB_FILE = os.path.join("data", "pos.db")

def get_connection():
    """Return a new database connection."""
    os.makedirs("data", exist_ok=True)
    conn = sqlite3.connect(DB_FILE)
    conn.row_factory = sqlite3.Row    # rows behave like dicts
    conn.execute("PRAGMA foreign_keys = ON")
    return conn
```

What this does, line by line:

- **os.makedirs("data", exist_ok=True)** — creates the *data* folder if it does not exist yet.
- **sqlite3.connect(DB_FILE)** — opens (or creates) the *pos.db* file.
- **conn.row_factory = sqlite3.Row** — lets us access columns by name (for example *row["price"]*).
- **PRAGMA foreign_keys = ON** — turns on foreign key enforcement (off by default in SQLite).

5.2 Creating the tables

Now add this *init_db()* function below *get_connection()*. It creates all four tables on the first run and inserts a default admin user so we can log in.

```
def init_db():
    """Create all tables if they don't exist yet."""
    conn = get_connection()
    c = conn.cursor()

    c.execute("""
        CREATE TABLE IF NOT EXISTS users (
            id         INTEGER PRIMARY KEY AUTOINCREMENT,
            username   TEXT UNIQUE NOT NULL,
            password   TEXT NOT NULL,
            role       TEXT DEFAULT 'cashier',
            full_name  TEXT
        )
    """)

    c.execute("""
        CREATE TABLE IF NOT EXISTS products (
            id        INTEGER PRIMARY KEY AUTOINCREMENT,
            code      TEXT UNIQUE NOT NULL,
            name      TEXT NOT NULL,
            price     REAL NOT NULL,
            stock     INTEGER DEFAULT 0,
            category  TEXT
        )
    """)

    c.execute("""
        CREATE TABLE IF NOT EXISTS sales (
            id          INTEGER PRIMARY KEY AUTOINCREMENT,
            bill_no     TEXT UNIQUE NOT NULL,
            total       REAL NOT NULL,
            cashier_id  INTEGER,
            sale_date   DATETIME DEFAULT CURRENT_TIMESTAMP,
            FOREIGN KEY (cashier_id) REFERENCES users(id)
        )
    """)

    c.execute("""
        CREATE TABLE IF NOT EXISTS sale_items (
            id          INTEGER PRIMARY KEY AUTOINCREMENT,
            sale_id     INTEGER NOT NULL,
            product_id  INTEGER NOT NULL,
            quantity    INTEGER NOT NULL,
            unit_price  REAL NOT NULL,
            subtotal    REAL NOT NULL,
            FOREIGN KEY (sale_id)    REFERENCES sales(id),
            FOREIGN KEY (product_id) REFERENCES products(id)
        )
    """)
```

```
    # Default admin user (password: admin)
    c.execute("SELECT COUNT(*) FROM users")
    if c.fetchone()[0] == 0:
        c.execute(
            "INSERT INTO users (username, password, role, full_name) "
            "VALUES (?, ?, ?, ?)",
            ("admin", "admin", "admin", "Administrator")
        )
    conn.commit()
    conn.close()
```

NOTE: *In real production, never store plain-text passwords. We use plain text here for simplicity. In Chapter 12 we discuss switching to hashlib or bcrypt.*

5.3 Product functions (CRUD)

CRUD stands for Create, Read, Update, Delete. Every business application needs these four operations for every table.

Below are the four product functions. Add them to *database.py* after *init_db()*.

```
def add_product(code, name, price, stock=0, category=""):
    conn = get_connection()
    conn.execute(
        "INSERT INTO products (code, name, price, stock, category) "
        "VALUES (?, ?, ?, ?, ?)",
        (code, name, price, stock, category)
    )
    conn.commit()
    conn.close()

def get_all_products():
    conn = get_connection()
    rows = conn.execute(
        "SELECT * FROM products ORDER BY name"
    ).fetchall()
    conn.close()
    return rows

def update_product(pid, code, name, price, stock, category):
    conn = get_connection()
    conn.execute(
        "UPDATE products SET code=?, name=?, price=?, "
        "stock=?, category=? WHERE id=?",
        (code, name, price, stock, category, pid)
    )
    conn.commit()
    conn.close()

def delete_product(pid):
```

```
    conn = get_connection()
    conn.execute("DELETE FROM products WHERE id=?", (pid,))
    conn.commit()
    conn.close()
```

Important things to notice in the code above:

- We use **?** placeholders, never f-strings, when inserting user data. This protects against **SQL injection** attacks.
- Every function opens a connection and closes it. SQLite is happy with this pattern for a desktop app.
- Functions return Python objects (Row), so the rest of our code never sees raw SQL.

5.4 Sales functions

The most important function in the whole database module is *save_sale()*. It inserts the sale header, all the sale lines, and reduces stock — all inside one transaction.

```
def save_sale(bill_no, total, cashier_id, items):
    """
    items = list of dicts:
       [{"product_id": 1, "qty": 2, "price": 350}, ...]
    """
    conn = get_connection()
    c = conn.cursor()
    c.execute(
        "INSERT INTO sales (bill_no, total, cashier_id) "
        "VALUES (?, ?, ?)",
        (bill_no, total, cashier_id)
    )
    sale_id = c.lastrowid
    for it in items:
        subtotal = it["qty"] * it["price"]
        c.execute(
            "INSERT INTO sale_items "
            "(sale_id, product_id, quantity, unit_price, subtotal) "
            "VALUES (?, ?, ?, ?, ?)",
            (sale_id, it["product_id"], it["qty"], it["price"], subtotal)
        )
        # Reduce stock
        c.execute(
            "UPDATE products SET stock = stock - ? WHERE id = ?",
            (it["qty"], it["product_id"])
        )
    conn.commit()
    conn.close()
    return sale_id
```

save_sale does three writes inside one transaction: it inserts into sales, inserts each line into sale_items, and reduces stock.

If anything fails, nothing is saved — that is the magic of *conn.commit()* at the end. SQLite holds all changes in memory until commit is called.

Daily sales report function

A simple function to fetch all sales for a given day:

```
def get_daily_sales(date_str):
    """date_str format: YYYY-MM-DD"""
    conn = get_connection()
    rows = conn.execute(
        "SELECT * FROM sales WHERE DATE(sale_date) = ? "
        "ORDER BY sale_date DESC", (date_str,)
    ).fetchall()
    conn.close()
    return rows
```

5.5 Login function

Finally, a small function to check if a username and password match a row in the users table.

```
def login(username, password):
    conn = get_connection()
    row = conn.execute(
        "SELECT * FROM users WHERE username=? AND password=?",
        (username, password)
    ).fetchone()
    conn.close()
    return row     # None if no match
```

That completes **database.py**. In the next chapter, we build the first window: the login screen.

Chapter 6 — The Login Window

Now we build our first screen using **Tkinter**, Python's built-in GUI library. Every window is built from **widgets** — labels, entry boxes, buttons.

We arrange widgets on the screen with one of three layout managers: *pack*, *grid*, or *place*. We will mostly use **grid** because it is the easiest to control.

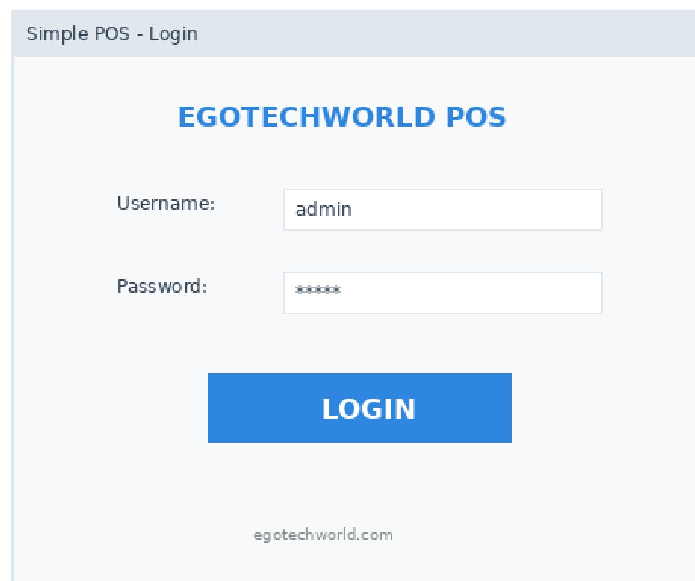

Figure 6.1 — The login window we will build

6.1 Create ui/login_window.py

Inside the *ui/* folder, create a new file named **login_window.py** and paste the code below.

```
import tkinter as tk
from tkinter import messagebox
import database as db

class LoginWindow:
    def __init__(self, on_success):
        self.on_success = on_success
        self.root = tk.Tk()
        self.root.title("Simple POS - Login")
        self.root.geometry("400x320")
        self.root.configure(bg="#F8F9FA")
        self.root.resizable(False, False)
        self._build()

    def _build(self):
```

```
        tk.Label(self.root, text="EGOTECHWORLD POS",
                 font=("Helvetica", 18, "bold"),
                 bg="#F8F9FA", fg="#2E86DE").pack(pady=20)

        frm = tk.Frame(self.root, bg="#F8F9FA")
        frm.pack(pady=10)

        tk.Label(frm, text="Username:", bg="#F8F9FA",
                 font=("Helvetica", 11)).grid(
            row=0, column=0, sticky="e", pady=8, padx=6)
        self.user_entry = tk.Entry(
            frm, font=("Helvetica", 11), width=22)
        self.user_entry.grid(row=0, column=1, pady=8)

        tk.Label(frm, text="Password:", bg="#F8F9FA",
                 font=("Helvetica", 11)).grid(
            row=1, column=0, sticky="e", pady=8, padx=6)
        self.pass_entry = tk.Entry(
            frm, font=("Helvetica", 11), width=22, show="*")
        self.pass_entry.grid(row=1, column=1, pady=8)

        tk.Button(self.root, text="LOGIN",
                  font=("Helvetica", 12, "bold"),
                  bg="#2E86DE", fg="white", width=20,
                  command=self.do_login).pack(pady=20)

        tk.Label(self.root, text="egotechworld.com",
                 bg="#F8F9FA", fg="#7F8C8D",
                 font=("Helvetica", 9)).pack(side="bottom", pady=10)

        self.root.bind("<Return>", lambda e: self.do_login())

    def do_login(self):
        u = self.user_entry.get().strip()
        p = self.pass_entry.get().strip()
        if not u or not p:
            messagebox.showerror("Error", "Please fill both fields")
            return
        user = db.login(u, p)
        if user:
            self.root.destroy()
            self.on_success(dict(user))
        else:
            messagebox.showerror(
                "Login Failed", "Invalid username or password")

    def run(self):
        self.root.mainloop()
```

6.2 Key Tkinter ideas in this code

- **tk.Tk()** — creates the actual window.
- **geometry("400x320")** — sets size in pixels.
- **configure(bg="...")** — sets the background color (clean light gray).
- **pack** places things one below the other; **grid** uses rows and columns. We use both in this window.
- **command=self.do_login** — what happens when the LOGIN button is clicked.
- **show="*"** — hides the password as dots while typing.
- **self.root.bind("<Return>", ...)** — pressing Enter submits the form, just like clicking LOGIN.

> ***NOTE:*** *The on_success parameter is a callback function. When login succeeds, we call it with the user data. This decouples the login window from whatever screen comes next — flexible and clean.*

Chapter 7 — The Main POS Window

This is the heart of the application — where cashiers spend most of their day.

On the left we show the product list, on the right the cart. The total updates live as the cashier adds items.

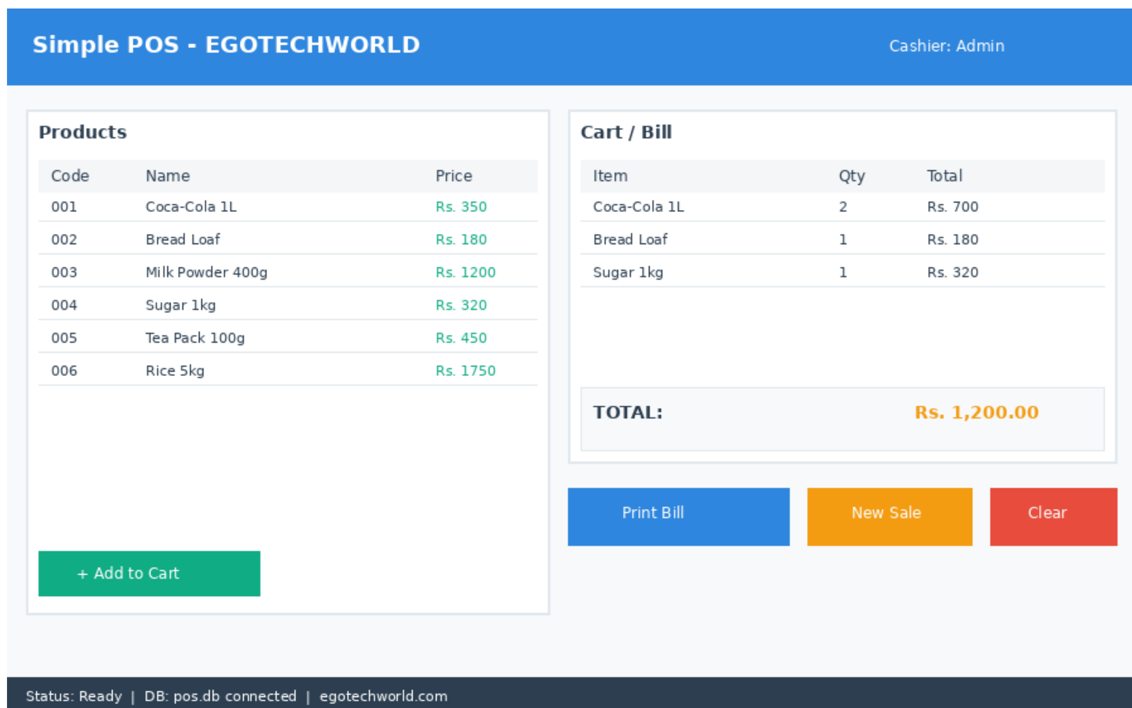

Figure 7.1 — Layout of the main POS window

7.1 Setting up the class

Create the file **ui/pos_window.py**. Start with the imports and the class constructor:

```
import tkinter as tk
from tkinter import ttk, messagebox
from datetime import datetime
import database as db

class POSWindow:
    def __init__(self, user):
        self.user = user
        self.cart = []       # list of dicts
        self.root = tk.Tk()
        self.root.title(
            f"Simple POS - Cashier: {user['full_name']}")
        self.root.geometry("1000x600")
        self.root.configure(bg="#F8F9FA")
```

```
        self._build()
        self._load_products()
```

The *self.cart* list will hold the items the cashier has added. Each item is a dictionary with product_id, name, qty, and price.

7.2 Building the two-column layout

The layout has three areas: a blue header bar at the top, a product panel on the left, and a cart panel on the right. Add this method to the class:

```
    def _build(self):
        # Header
        header = tk.Frame(self.root, bg="#2E86DE", height=50)
        header.pack(fill="x")
        tk.Label(
            header, text="Simple POS - EGOTECHWORLD",
            bg="#2E86DE", fg="white",
            font=("Helvetica", 14, "bold")
        ).pack(side="left", padx=20, pady=12)
        tk.Label(
            header,
            text=f"Cashier: {self.user['full_name']}",
            bg="#2E86DE", fg="white",
            font=("Helvetica", 11)
        ).pack(side="right", padx=20)

        # Body - two columns
        body = tk.Frame(self.root, bg="#F8F9FA")
        body.pack(fill="both", expand=True, padx=10, pady=10)

        # Left: products panel
        left = tk.LabelFrame(
            body, text="Products", bg="white",
            font=("Helvetica", 11, "bold"))
        left.pack(side="left", fill="both", expand=True,
                  padx=(0, 5))

        cols = ("code", "name", "price", "stock")
        self.prod_tree = ttk.Treeview(
            left, columns=cols, show="headings", height=18)
        for c, w in zip(cols, (70, 220, 80, 60)):
            self.prod_tree.heading(c, text=c.title())
            self.prod_tree.column(c, width=w)
        self.prod_tree.pack(
            fill="both", expand=True, padx=8, pady=8)

        tk.Button(
            left, text="+ Add to Cart",
            bg="#10AC84", fg="white",
            font=("Helvetica", 11, "bold"),
```

```
            command=self.add_to_cart
        ).pack(pady=(0, 8))
```

Now continue the same *_build* method with the right-side cart panel and the action buttons:

```
        # Right: cart panel
        right = tk.LabelFrame(
            body, text="Cart / Bill", bg="white",
            font=("Helvetica", 11, "bold"))
        right.pack(side="right", fill="both", expand=True,
                   padx=(5, 0))

        cart_cols = ("name", "qty", "price", "subtotal")
        self.cart_tree = ttk.Treeview(
            right, columns=cart_cols,
            show="headings", height=14)
        for c, w in zip(cart_cols, (200, 60, 90, 100)):
            self.cart_tree.heading(c, text=c.title())
            self.cart_tree.column(c, width=w)
        self.cart_tree.pack(
            fill="both", expand=True, padx=8, pady=8)

        # Total label
        self.total_var = tk.StringVar(value="TOTAL: Rs. 0.00")
        tk.Label(
            right, textvariable=self.total_var,
            font=("Helvetica", 16, "bold"),
            bg="white", fg="#F39C12"
        ).pack(pady=10)

        # Action buttons
        btn_row = tk.Frame(right, bg="white")
        btn_row.pack(pady=(0, 10))
        tk.Button(
            btn_row, text="Print Bill",
            bg="#2E86DE", fg="white",
            font=("Helvetica", 11, "bold"), width=12,
            command=self.checkout
        ).pack(side="left", padx=4)
        tk.Button(
            btn_row, text="Clear",
            bg="#E74C3C", fg="white",
            font=("Helvetica", 11, "bold"), width=10,
            command=self.clear_cart
        ).pack(side="left", padx=4)
```

7.3 Loading products and adding to cart

These two methods load products from the database and let the cashier add a selected product to the cart. If the same product is added again, the quantity simply increases instead of creating a new line.

```
def _load_products(self):
    for r in self.prod_tree.get_children():
        self.prod_tree.delete(r)
    for p in db.get_all_products():
        self.prod_tree.insert(
            "", "end",
            values=(p["code"], p["name"],
                    f"{p['price']:.2f}", p["stock"]),
            iid=str(p["id"]))

def add_to_cart(self):
    sel = self.prod_tree.selection()
    if not sel:
        messagebox.showwarning(
            "Pick one", "Select a product first")
        return
    pid = int(sel[0])
    vals = self.prod_tree.item(sel[0], "values")
    # check if already in cart
    for it in self.cart:
        if it["product_id"] == pid:
            it["qty"] += 1
            self._refresh_cart()
            return
    self.cart.append({
        "product_id": pid,
        "name":  vals[1],
        "qty":   1,
        "price": float(vals[2]),
    })
    self._refresh_cart()
```

The *_refresh_cart* method updates the cart Treeview and recalculates the total whenever an item is added or removed:

```
def _refresh_cart(self):
    for r in self.cart_tree.get_children():
        self.cart_tree.delete(r)
    total = 0
    for it in self.cart:
        sub = it["qty"] * it["price"]
        total += sub
        self.cart_tree.insert(
            "", "end",
            values=(it["name"], it["qty"],
                    f"{it['price']:.2f}",
                    f"{sub:.2f}"))
```

```
        self.total_var.set(f"TOTAL: Rs. {total:.2f}")
```

7.4 Checkout — saving the sale

When the cashier clicks "Print Bill", we save the sale to the database, show the receipt, clear the cart, and reload the products (because the stock numbers have changed).

```
    def checkout(self):
        if not self.cart:
            messagebox.showwarning("Empty", "Cart is empty")
            return
        total = sum(it["qty"] * it["price"] for it in self.cart)
        bill_no = "INV-" + datetime.now().strftime(
            "%Y%m%d-%H%M%S")
        db.save_sale(
            bill_no, total, self.user["id"], self.cart)
        self._print_bill(bill_no, total)
        self.cart.clear()
        self._refresh_cart()
        self._load_products()       # stock changed
```

7.5 Printing the receipt

The receipt is shown in a new *Toplevel* window using a monospace font so the columns line up. This is the same layout a real thermal printer would use.

Figure 7.2 — A sample receipt window

```
def _print_bill(self, bill_no, total):
    win = tk.Toplevel(self.root)
    win.title("Receipt")
    win.geometry("360x500")
    win.configure(bg="white")
    text = tk.Text(
        win, font=("Courier", 10),
        bg="white", bd=0)
    text.pack(fill="both", expand=True, padx=10, pady=10)
    lines = []
    lines.append("==================================")
    lines.append("       EGOTECHWORLD STORE")
    lines.append("       egotechworld.com")
    lines.append("==================================")
    lines.append(f"Bill: {bill_no}")
    lines.append(
        f"Date: {datetime.now():%Y-%m-%d %H:%M}")
    lines.append(f"Cashier: {self.user['full_name']}")
    lines.append("----------------------------------")
    for it in self.cart:
        sub = it["qty"] * it["price"]
        lines.append(
            f"{it['name'][:18]:<18} "
            f"{it['qty']:>3} x {it['price']:>7.2f}")
        lines.append(f"{'':>20}{sub:>12.2f}")
    lines.append("----------------------------------")
    lines.append(f"TOTAL: Rs. {total:>20.2f}")
    lines.append("==================================")
    lines.append("    Thank you, come again!")
```

```
        text.insert("1.0", "\n".join(lines))
        text.config(state="disabled")

    def clear_cart(self):
        self.cart.clear()
        self._refresh_cart()

    def run(self):
        self.root.mainloop()
```

That completes *pos_window.py*. In the next chapter we wire everything together with a tiny *main.py* and run the app for the first time.

Chapter 8 — Wiring it all together

The **main.py** file is the entry point. It initializes the database, shows the login window, and on a successful login opens the POS window.

8.1 Create main.py

Open **main.py** at the project root and paste:

```
import database as db
from ui.login_window import LoginWindow
from ui.pos_window import POSWindow

def open_pos(user):
    POSWindow(user).run()

def main():
    db.init_db()                  # create tables on first run
    LoginWindow(on_success=open_pos).run()

if __name__ == "__main__":
    main()
```

Now run your app from the terminal:

```
cd simple_pos
python main.py
```

A login window appears. Type **admin** / **admin** and click LOGIN. The main POS window opens. It will be empty for now — head to Section 8.2 to add some sample products.

8.2 Adding sample products quickly

Until we build a product management screen, you can pre-load some products by running the small script below. Save it as **seed.py** at the project root.

```
import database as db

db.init_db()
samples = [
    ("P001", "Coca-Cola 1L",      350.00, 50, "Drinks"),
    ("P002", "Bread Loaf",        180.00, 30, "Bakery"),
    ("P003", "Milk Powder 400g", 1200.00, 20, "Dairy"),
    ("P004", "Sugar 1kg",          320.00, 40, "Grocery"),
    ("P005", "Tea Pack 100g",      450.00, 25, "Drinks"),
    ("P006", "Rice 5kg",          1750.00, 15, "Grocery"),
```

```
]
for s in samples:
    try:
        db.add_product(*s)
    except Exception as e:
        print("skip:", e)
print("Done")
```

Run it once:

```
python seed.py
```

Restart the POS app. You should now see all six products in the left panel. Try adding a few to the cart and clicking Print Bill — your first sale is complete!

> ***NOTE:*** *After your first sale, open data/pos.db with DB Browser for SQLite to see your data physically stored in tables. This is a great way to verify everything is working at the database level.*

Chapter 9 — Software Testing

Testing means writing small programs that automatically check your main code is working correctly.

You don't have to manually click around every time you change something — instead, the test suite tells you in 5 seconds whether anything broke. This is one of the most professional habits a developer can build.

9.1 Types of tests in our POS

- **Unit tests** — test one function at a time (e.g. *add_product()* works correctly).
- **Integration tests** — test that two modules work together (e.g. saving a sale also reduces stock).
- **Manual tests** — clicking through the GUI yourself to check the visual side.

9.2 Writing unit tests with unittest

Python comes with a built-in test framework called *unittest*. Create the file **tests/test_database.py** with the code below.

```
import unittest
import os, sys
sys.path.insert(0, os.path.abspath(
    os.path.join(os.path.dirname(__file__), "..")))

import database as db

class TestDatabase(unittest.TestCase):
    def setUp(self):
        # Use a fresh test DB for every test
        db.DB_FILE = "data/test.db"
        if os.path.exists(db.DB_FILE):
            os.remove(db.DB_FILE)
        db.init_db()

    def test_default_admin_exists(self):
        u = db.login("admin", "admin")
        self.assertIsNotNone(u)
        self.assertEqual(u["role"], "admin")

    def test_add_product(self):
        db.add_product("X1", "Test Item", 100.0, 5, "Test")
        items = db.get_all_products()
```

```
        self.assertEqual(len(items), 1)
        self.assertEqual(items[0]["name"], "Test Item")

    def test_save_sale_reduces_stock(self):
        db.add_product("X2", "Apple", 50.0, 10)
        prod = db.get_all_products()[0]
        db.save_sale("INV-T-1", 150.0, 1,
                     [{"product_id": prod["id"],
                       "qty": 3, "price": 50.0}])
        prod_after = db.get_all_products()[0]
        self.assertEqual(prod_after["stock"], 7)

if __name__ == "__main__":
    unittest.main()
```

Run all tests from the project root:

```
python -m unittest discover tests
```

You should see **OK** at the end. Each dot is one passing test. If you ever see **F** (failure) or **E** (error), the message tells you exactly which line broke. This is your safety net.

9.3 A simple manual test checklist

Automated tests cover the database side. For the visual side, click through this checklist yourself before delivering to a client.

#	Test	Expected Result
1	Open app, type wrong password	Error message appears
2	Login with admin/admin	POS window opens
3	Click product, click Add to Cart	Item appears in cart
4	Add same product again	Quantity increases by 1
5	Click Print Bill	Receipt window appears
6	Re-open product list	Stock has reduced by sold qty
7	Click Clear	Cart empties, total is 0
8	Close and reopen app	Old sales are still in DB

Chapter 10 — Packaging into a Setup File

Right now your customer would need Python installed and would have to run a script. That is not professional.

We turn the project into a real Windows installer in two steps:

- **Step 1** — Use **PyInstaller** to convert *main.py* into a single *SimplePOS.exe* file. Customers don't need Python on their computer.
- **Step 2** — Use **Inno Setup** to wrap that .exe and the data folder into a friendly *SimplePOS-Setup.exe* installer with a Start menu shortcut.

10.1 Step 1 — PyInstaller

Install PyInstaller (one time):

```
pip install pyinstaller
```

From the project folder, run:

```
pyinstaller --noconfirm --onefile --windowed ^
    --name SimplePOS ^
    --icon assets/logo.ico ^
    --add-data "data;data" ^
    main.py
```

Explanation of each flag:

- **--onefile** — bundle everything into one .exe.
- **--windowed** — hide the black command prompt.
- **--name SimplePOS** — output file is *SimplePOS.exe*.
- **--icon** — give your app a custom icon (optional).
- **--add-data** — include the data folder.

When done, look in the **dist/** folder. Double-click *SimplePOS.exe* — your app runs as a stand-alone program. You can copy this single file to any Windows computer and it will run.

***NOTE:** On macOS or Linux use a colon instead of semicolon (--add-data "data:data"). Only Windows uses semicolon.*

10.2 Step 2 — Inno Setup (Windows installer)

Inno Setup is a free tool that wraps your .exe into a real Windows installer — the kind users are used to (Next, Next, I Agree, Install, Finish). Download from *jrsoftware.org/isdl.php* and install it.

Create **setup_installer.iss** in your project folder:

```
[Setup]
AppName=Simple POS by EGOTECHWORLD
AppVersion=1.0
AppPublisher=EGOTECHWORLD PVT LTD
AppPublisherURL=https://egotechworld.com
DefaultDirName={pf}\SimplePOS
DefaultGroupName=Simple POS
OutputBaseFilename=SimplePOS-Setup
Compression=lzma
SolidCompression=yes
WizardStyle=modern

[Files]
Source: "dist\SimplePOS.exe"; DestDir: "{app}"; Flags: ignoreversion
Source: "data\*"; DestDir: "{app}\data"; Flags: ignoreversion recursesubdirs
Source: "README.md"; DestDir: "{app}"; Flags: ignoreversion

[Icons]
Name: "{group}\Simple POS"; Filename: "{app}\SimplePOS.exe"
Name: "{commondesktop}\Simple POS"; Filename: "{app}\SimplePOS.exe"

[Run]
Filename: "{app}\SimplePOS.exe"; Description: "Launch Simple POS"; Flags: nowait
postinstall skipifsilent
```

Then build the installer:

- Open the Inno Setup Compiler.
- File → Open → choose *setup_installer.iss*.
- Click the green Run button (or Build → Compile).
- Inno produces **SimplePOS-Setup.exe** in the *Output* folder.

That single file is now your product. Send it to a customer; they double-click, click through the wizard, and Simple POS appears in their Start menu and on the desktop.

Chapter 11 — Selling to a Client

You have a finished product. Now it is time to put money in your pocket. This chapter covers pricing, the demo, the deployment visit, and ongoing support.

11.1 Pricing your POS

For Sri Lankan small shops in 2026, the realistic price ranges are shown in the table below. Adjust for your local market and your experience level.

Package	What's included	Suggested Price (LKR)
Basic	Software install on 1 PC, 1 hour training	Rs. 12,000 - 18,000
Standard	Basic + custom logo + 30-day support	Rs. 20,000 - 30,000
Premium	Standard + barcode setup + 6-month support	Rs. 40,000 - 60,000
Source code	Full Python source for a developer	Rs. 8,000 - 15,000

NOTE: *Charge a small annual maintenance fee (10–15% of original price) for bug fixes, backups, and small changes. This becomes recurring income.*

11.2 The 5-step sales process

Selling software is mostly about solving real pain. Follow these five steps and you will close most deals:

- **1. Find a target shop** — small grocery, mini-mart, bakery, pharmacy. Owners who still use paper bill books are perfect candidates.
- **2. Free demo (15 minutes)** — open your laptop, log in, ring up a pretend sale, print the bill. Show them the daily sales report.
- **3. Ask about their pain** — "How do you know what you sold yesterday?" "How do you know which products are out of stock?" Let them complain. Your software solves those exact pains.
- **4. Quote** — give a one-page printed quotation with package name, price, and what is included. Always offer two packages so they choose between two options instead of yes/no.

- **5. Close** — ask for 50% advance to start work. Deliver in 1–3 days. Collect the rest after installation.

11.3 Installation visit checklist

Before going to the customer's shop, prepare these items:

- A USB stick with **SimplePOS-Setup.exe**.
- A printed copy of **README.md** with login details and basic steps.
- Your default admin password ready to change to a customer-chosen one.
- A blank notebook for noting the customer's product list.
- (Optional) A thermal receipt printer if the package includes one.

Once you arrive at the shop:

- Install the .exe, launch the app, change the default admin password.
- Add their first 10–20 products together with the owner. This trains them.
- Print a real test bill on their printer or screen.
- Show how to view yesterday's sales.
- Take a photo of the running app for your portfolio (with permission).
- Hand over the printed README and your phone number.

11.4 Backup — a small thing that prevents big problems

The whole database is just one file: *data/pos.db*. Tell the customer to copy this file to a USB or Google Drive once a week.

If their computer crashes, paste this file back and they have **every sale ever** intact.

For an extra Rs. 500/month, offer to do this backup automatically — a tiny Python script that emails the .db file to a Gmail account each night. Customers love this kind of "set and forget" service.

Chapter 12 — Where to go next

You now have a working, sellable POS. To grow it into a serious product, add these features one by one.

12.1 Easy additions (a few hours each)

- **Product management screen** — instead of using *seed.py*, build a window to add/edit/delete products with the mouse.
- **Bill discount** — a percentage discount field above the total.
- **Customer name on bill** — small entry above Print Bill.
- **Search products by code** — type the product code and press Enter to add directly to cart.
- **Hashed passwords** — replace plain text with *hashlib.sha256*.

12.2 Medium effort (one weekend each)

- **Barcode scanner support** — a USB scanner just types the barcode like a keyboard, so a search-by-code field already supports it.
- **Sales report screen** — pick a date, see total sales and top products.
- **Export to Excel** — using the *openpyxl* library.
- **Multi-user roles** — only admin can edit products; cashiers only sell.
- **Auto backup** — a scheduler that copies *pos.db* to a backup folder every day.

12.3 Advanced (long-term)

- **Switch SQLite to MySQL or PostgreSQL** — for multi-PC shops on a LAN.
- **Cloud sync** — push every sale to a remote server so the owner can see reports from home.
- **Mobile companion app** — the owner views sales from their phone.
- **Multi-shop / multi-branch** — group reports across several shops.
- **Receipt printer driver** — auto-print to a real thermal printer instead of just showing on screen.

12.4 Final words

A POS system is one of the most useful applications a junior developer can build. It is simple enough to finish in a few weekends, but valuable enough that real businesses will pay you for it.

Every feature you add is one more reason for a shop to choose your software over a competitor.

If you found this tutorial helpful, share it with another learner and visit **egotechworld.com** for more beginner-friendly tutorials, project ideas, and source code downloads.

We are building a complete library for self-taught developers in Sri Lanka and around the world.

Good luck — and may your first POS sale be the first of many.

EGOTECHWORLD PVT LTD
A Sri Lankan IT company building tools, tutorials, and source code for self-taught developers worldwide.

Tutorials • Source Code • Coding Courses • AI Tools

egotechworld.com