Python Development Setup with IntelliJ IDEA

Introduction

IntelliJ IDEA, primarily known for Java development, also provides robust support for Python when used with the Python plugin. It offers a range of features that enhance productivity and streamline development. These include:

- Dedicated module type
- Ability to configure interpreters
- Python console
- Run/debug configurations for Python, and Python remote debug
- Intelligent code completion and navigation
- Built-in code inspections and quick fixes
- Integrated terminal and version control support
- Easy installation and management of external libraries

Purpose

This document outlines the steps to install IntelliJ IDEA, configure it for Python development using the Python plugin, run a sample Python script, and install external libraries using pip.

1. **Prerequisites**

- Operating System : Windows , macOS or Linux
- Installed : Python(if not download and install python first)

2. **Step_by_Step Installation of Python**

- Go to the official Python website: python.org/downloads/.
- Download the appropriate installer for your operating system (Windows, macOS, or Linux). For Windows, you'll typically download a .exe file.
- Locate the downloaded installer file (usually in your Downloads folder) and run it.
- On Windows, you'll see an option to "Add Python to PATH". Make sure to check this box to easily access Python from your command line.
- Follow the on-screen instructions to finish the installation process. This may involve clicking "Install Now" and waiting for the process to complete.

3. **Verify Python installation**

- Open your command prompt or terminal.
- Type python --version (or python -V) and press Enter.
- If Python is installed correctly, you should see the Python version number displayed.

4. **Step_by_Step_Installation of Intellij IDEA**

- Visit https://www.jetbrains.com/idea/download/
- Download the Community Edition (free version)
- Install using default settings

Python Development Setup with IntelliJ IDEA

- Launch IntelliJ IDEA

5. Configure Intellij for Python

- Install Python Plugin
- Open IntelliJ IDEA.
- Go to File > Settings > Plugins (or IntelliJ IDEA > Preferences > Plugins on macOS)
- Search for Python plugin
- Click Install and restart Intellij when prompted

6. Create and Run a Python Project

- Go to File > New > Project
- Select Python on the left sidebar
- Choose a virtual environment or system interpreter
- Click "..." to locate your Python executable (e.g., C:\Python311\python.exe)
- Click Create

7. Create Python File

- Right-click src or project folder > New > Python File
- Name it like hello.py
- Add python code(eg : print("Hello Intellij")

8. Run Python File

- Right-click on hello.py
- Select **Run**
- Output displayed

9. Steps to Install Python Libraries

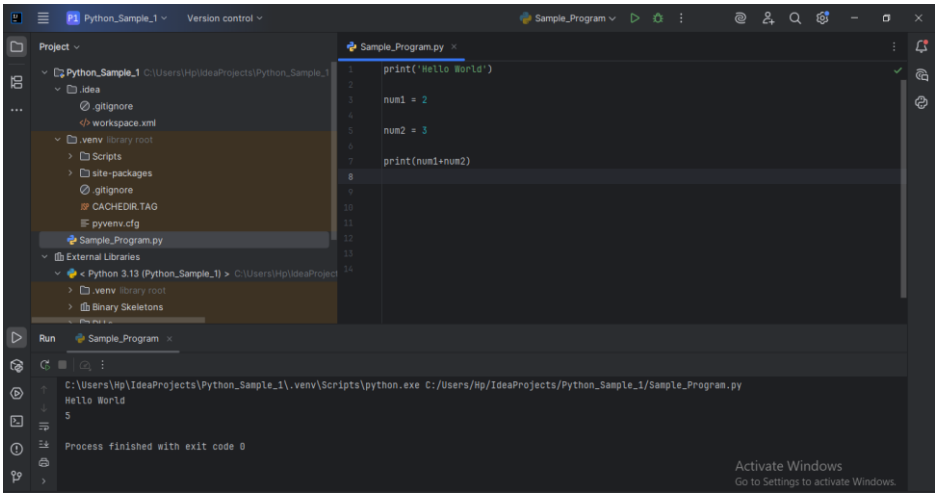
- Open or Create a Python Project
- Configure Your Python Interpreter(Go to File → Settings → Project: <your_project> → Python Interpreter)
- If no interpreter is configured, click the gear icon → Add → Select: Virtualenv, Conda, or System Interpreter
- Click OK to apply
- In the same Python Interpreter window, you'll see a list of installed packages.
- Click the ➕ Add button (bottom left)
- In the dialog, type the name of the library you want (e.g., requests, numpy)
- Click Install Package
- IntelliJ will show progress, and notify you when installation completes

10. (Alternative) Install via Terminal

- Go to View → Tool Windows → Terminal (or press Alt + F12)
- Pip install packages

Image of Intellij IDEA with Python code as shown below :

Python Development Setup with IntelliJ IDEA



Basics of Python Coding

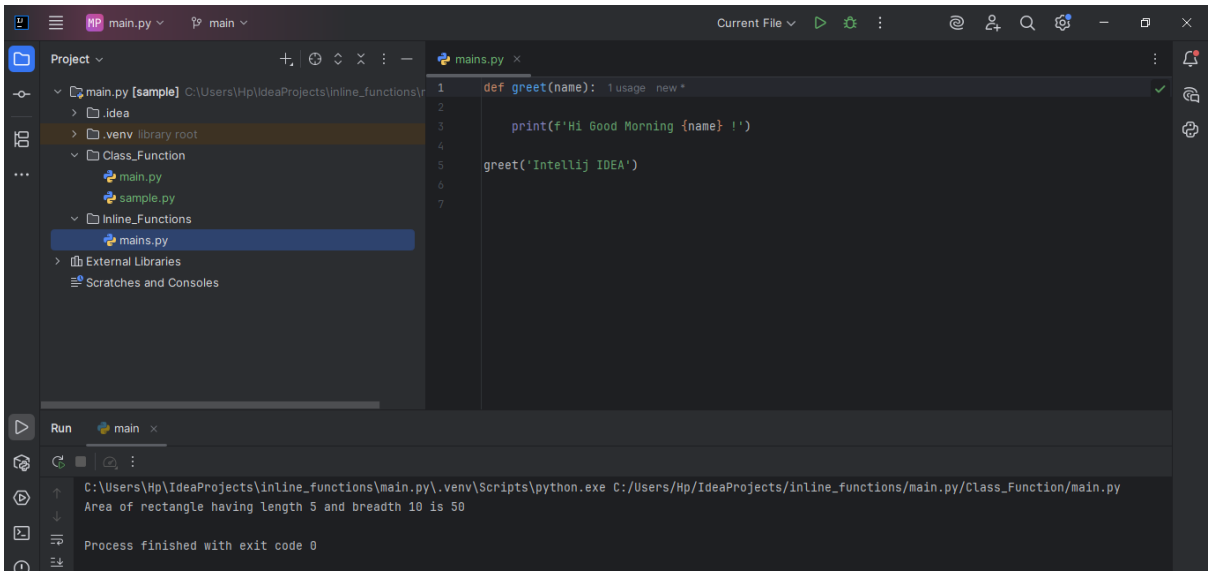
Two core Python concepts using IntelliJ IDEA as :

1. Inline function in the same file
2. Class and method in a separate module

1. Inline function in the same file

- Open IntelliJ IDEA and create a Python project named ‘python_works’
- Right-click the root folder → New > Directory → name it ‘Inline_Functions’
- Right-click the new folder → New > Python File → name it ‘mains.py’
- Obtain the sample welcome code and Run

Figure below shows code with inline function and its output ;



2. Class and method in a separate module

- Right-click the project root → New > Directory → name it ‘Class_Function’
- Right-click Class_Function → New > Python File → name it ‘sample.py’
- Obtain the sample code for finding area of rectangle and Run
- Create another Python file named ‘main.py’ in the same folder
- In ‘main.py’ import class from ‘sample.py’ and Run

Figure below shows sample.py file ;

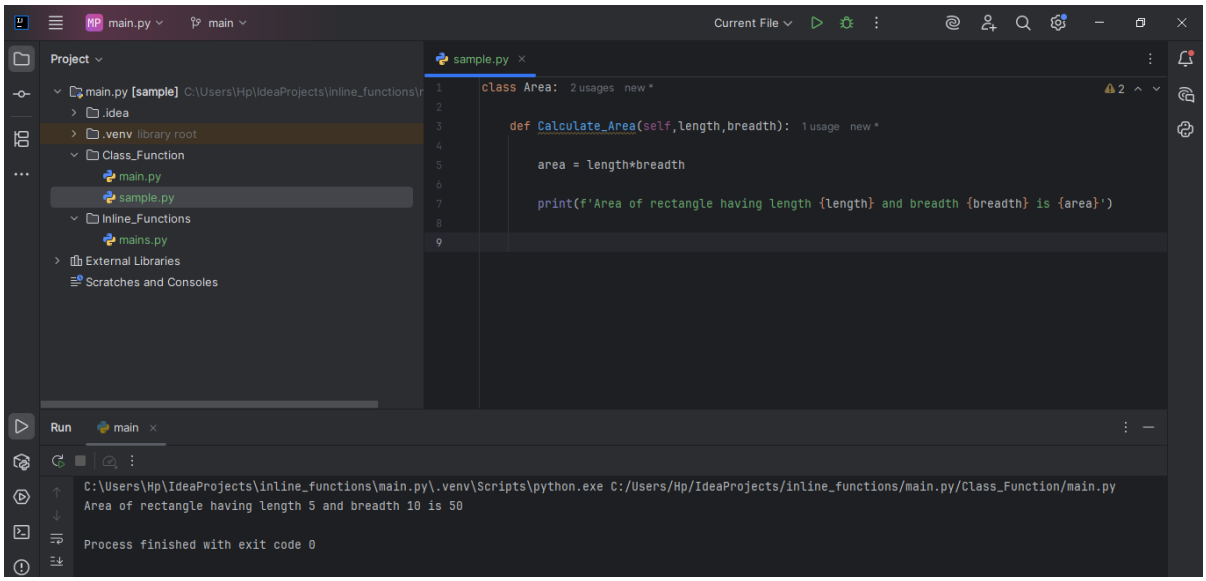
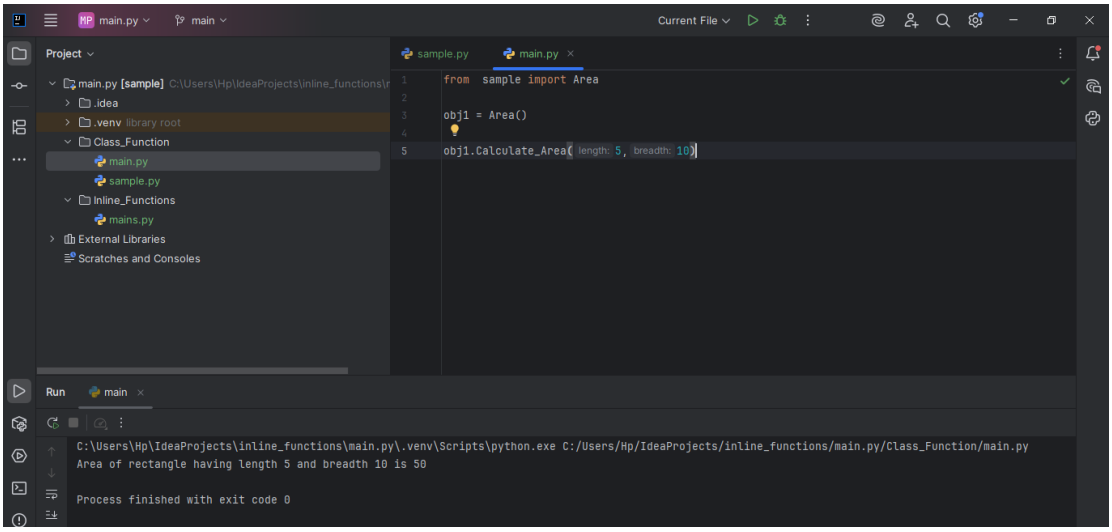


Figure below shows main.py file ;



Creating GUI's to process data from Excel Sheet

Creating a GUI in Python using Tkinter to process data from an Excel sheet involves several key steps:

- Import Necessary Libraries
- Initialize the Tkinter Window
- Create GUI Elements
- Implement Functions for Excel Interaction
- Connect GUI Elements to Functions
- Run the Tkinter Event Loop

Python code for GUI to process Excel is as shown below :

```
import tkinter as tk
from tkinter import ttk, filedialog, messagebox
import pandas as pd
import numpy as np
import os

from matplotlib.backends.backend_tkagg import FigureCanvasTkAgg
from matplotlib.figure import Figure
# Utility: Table display in Treeview

def clear_tree(tree):
    tree.delete(*tree.get_children())
    for col in tree["columns"]:
        tree.heading(col, text="")
    tree["columns"] = ()

def show_dataframe_in_tree(tree, df: pd.DataFrame, max_rows=500):
    clear_tree(tree)
    if df is None or df.empty:
        return
    # Convert index to a column for display if it's not default RangeIndex
    df_to_show = df.copy()
    if not isinstance(df_to_show.index, pd.RangeIndex):
        df_to_show.insert(0, "(index)", df_to_show.index)
```

```
    cols = list(df_to_show.columns)
    tree["columns"] = cols
    tree["show"] = "headings"
    for col in cols:
        tree.heading(col, text=col)
        tree.column(col, width=140, stretch=True, anchor=tk.W)

    for _, row in df_to_show.head(max_rows).iterrows():
        tree.insert("", tk.END, values=[row.get(c, "") for c in cols])

# Main App
class ExcelProcessorApp(tk.Tk):
    def __init__(self):
        super().__init__()
        self.title("Excel Data Processor - TPM Toolkit")
        self.geometry("1200x750")

        self.df = None           # original DataFrame
        self.current = None      # current result DataFrame (from ops)
        self.last_plot_canvas = None
        self._build_ui()

    def _build_ui(self):
        # Top controls: file + sheet
        top = ttk.Frame(self)
        top.pack(fill="x", padx=10, pady=10)
        self.file_var = tk.StringVar()
        ttk.Label(top, text="Excel file:").pack(side="left")
        ttk.Entry(top, textvariable=self.file_var, width=60).pack(side="left",
padx=5)
        ttk.Button(top, text="Browse",
command=self.browse_file).pack(side="left", padx=5)

        ttk.Label(top, text="Sheet:").pack(side="left", padx=(15, 3))
        self.sheet_cmb = ttk.Combobox(top, width=25, state="readonly")
        self.sheet_cmb.pack(side="left")
        ttk.Button(top, text="Load", command=self.load_excel).pack(side="left",
padx=5)
```

```
        # Notebook for Operations
        self.nb = ttk.Notebook(self)
        self.nb.pack(fill="both", expand=True, padx=10, pady=(0,10))

        # Tabs
        self.tab_preview = ttk.Frame(self.nb)
        self.tab_summary = ttk.Frame(self.nb)
        self.tab_filter = ttk.Frame(self.nb)
        self.tab_pivot = ttk.Frame(self.nb)
        self.tab_chart = ttk.Frame(self.nb)

        self.nb.add(self.tab_preview, text="Preview")
        self.nb.add(self.tab_summary, text="Summary")
        self.nb.add(self.tab_filter, text="Filter")
        self.nb.add(self.tab_pivot, text="Pivot")
        self.nb.add(self.tab_chart, text="Chart")

        # Preview
        self.tree_preview = ttk.Treeview(self.tab_preview)
        self.tree_preview.pack(fill="both", expand=True)
        pv_btns = ttk.Frame(self.tab_preview)
        pv_btns.pack(fill="x")
        ttk.Button(pv_btns, text="Use Preview as Current",
command=self.use_preview_as_current).pack(side="left", padx=5, pady=5)

        # Summary tab
        self._build_summary_tab()

        # Filter tab
        self._build_filter_tab()

        # Pivot tab
        self._build_pivot_tab()

        # Chart tab
        self._build_chart_tab()

        # Bottom bar
        bottom = ttk.Frame(self)
```

```
        bottom.pack(fill="x", padx=10, pady=10)
        ttk.Button(bottom, text="Export Current to Excel",
command=self.export_current).pack(side="left", padx=5)
        ttk.Button(bottom, text="Reset (Reload File)",
command=self.reset_current).pack(side="left", padx=5)

        self.status_var = tk.StringVar(value="Load an Excel file to begin.")
        ttk.Label(self, textvariable=self.status_var, anchor="w").pack(fill="x",
padx=10, pady=(0,10))

    # -------- Tab Builders --------
    def _build_summary_tab(self):
        frm = ttk.Frame(self.tab_summary)
        frm.pack(fill="x", padx=10, pady=10)

        ttk.Label(frm, text="Numeric columns:").grid(row=0, column=0,
sticky="w")
        self.summary_num_cols = tk.Listbox(frm, selectmode="extended",
height=6, exportselection=False)
        self.summary_num_cols.grid(row=1, column=0, sticky="nsew",
padx=(0,10))

        ttk.Label(frm, text="Group by (optional):").grid(row=0, column=1,
sticky="w")
        self.summary_group = ttk.Combobox(frm, state="readonly")
        self.summary_group.grid(row=1, column=1, sticky="ew")

        ttk.Label(frm, text="Aggregations:").grid(row=0, column=2, sticky="w")
        self.agg_mean = tk.BooleanVar(value=True)
        self.agg_std = tk.BooleanVar(value=True)
        self.agg_min = tk.BooleanVar(value=False)
        self.agg_max = tk.BooleanVar(value=False)
        agg_box = ttk.Frame(frm)
        agg_box.grid(row=1, column=2, sticky="nw")
        ttk.Checkbutton(agg_box, text="mean",
variable=self.agg_mean).pack(anchor="w")
        ttk.Checkbutton(agg_box, text="std",
variable=self.agg_std).pack(anchor="w")
        ttk.Checkbutton(agg_box, text="min",
```

```
variable=self.agg_min).pack(anchor="w")
        ttk.Checkbutton(agg_box, text="max",
variable=self.agg_max).pack(anchor="w")

        ttk.Button(frm, text="Compute Summary",
command=self.compute_summary).grid(row=1, column=3, padx=10,
sticky="n")

        # results table
        self.tree_summary = ttk.Treeview(self.tab_summary)
        self.tree_summary.pack(fill="both", expand=True, padx=10, pady=(0,10))

        for i in range(3):
            frm.columnconfigure(i, weight=1)

    def _build_filter_tab(self):
        frm = ttk.Frame(self.tab_filter)
        frm.pack(fill="x", padx=10, pady=10)

        ttk.Label(frm, text="Column").grid(row=0, column=0, sticky="w")
        self.filter_col = ttk.Combobox(frm, state="readonly", width=30)
        self.filter_col.grid(row=1, column=0, sticky="ew", padx=(0,10))

        ttk.Label(frm, text="Operator").grid(row=0, column=1, sticky="w")
        self.filter_op = ttk.Combobox(frm, state="readonly", width=20,
                                      values=["== (equals)", "!= (not equals)", "contains", ">",
">=", "<", "<="])
        self.filter_op.grid(row=1, column=1, sticky="ew", padx=(0,10))
        self.filter_op.current(0)

        ttk.Label(frm, text="Value").grid(row=0, column=2, sticky="w")
        self.filter_val = ttk.Entry(frm, width=30)
        self.filter_val.grid(row=1, column=2, sticky="ew", padx=(0,10))

        ttk.Button(frm, text="Apply Filter (on Current or Original)",
command=self.apply_filter).grid(row=1, column=3, sticky="w")

        self.tree_filter = ttk.Treeview(self.tab_filter)
        self.tree_filter.pack(fill="both", expand=True, padx=10, pady=(0,10))
```

```
        for i in range(3):
            frm.columnconfigure(i, weight=1)

    def _build_pivot_tab(self):
        frm = ttk.Frame(self.tab_pivot)
        frm.pack(fill="x", padx=10, pady=10)

        ttk.Label(frm, text="Index").grid(row=0, column=0, sticky="w")
        self.pivot_index = ttk.Combobox(frm, state="readonly", width=25)
        self.pivot_index.grid(row=1, column=0, sticky="ew", padx=(0,10))

        ttk.Label(frm, text="Columns").grid(row=0, column=1, sticky="w")
        self.pivot_columns = ttk.Combobox(frm, state="readonly", width=25)
        self.pivot_columns.grid(row=1, column=1, sticky="ew", padx=(0,10))

        ttk.Label(frm, text="Values").grid(row=0, column=2, sticky="w")
        self.pivot_values = ttk.Combobox(frm, state="readonly", width=25)
        self.pivot_values.grid(row=1, column=2, sticky="ew", padx=(0,10))

        ttk.Label(frm, text="Agg").grid(row=0, column=3, sticky="w")
        self.pivot_agg = ttk.Combobox(frm, state="readonly", width=12,
                                      values=["mean", "sum", "count", "min", "max",
"median", "std"])
        self.pivot_agg.grid(row=1, column=3, sticky="ew", padx=(0,10))
        self.pivot_agg.current(0)

        ttk.Button(frm, text="Build Pivot",
command=self.build_pivot).grid(row=1, column=4, sticky="w")

        self.tree_pivot = ttk.Treeview(self.tab_pivot)
        self.tree_pivot.pack(fill="both", expand=True, padx=10, pady=(0,10))

        for i in range(4):
            frm.columnconfigure(i, weight=1)

    def _build_chart_tab(self):
        frm = ttk.Frame(self.tab_chart)
        frm.pack(fill="x", padx=10, pady=10)
```

```
        ttk.Label(frm, text="Data Source").grid(row=0, column=0, sticky="w")
        self.chart_source = ttk.Combobox(frm, state="readonly", values=["Current
result", "Original data"])
        self.chart_source.grid(row=1, column=0, sticky="ew", padx=(0,10))
        self.chart_source.current(0)

        ttk.Label(frm, text="X axis").grid(row=0, column=1, sticky="w")
        self.chart_x = ttk.Combobox(frm, state="readonly", width=25)
        self.chart_x.grid(row=1, column=1, sticky="ew", padx=(0,10))

        ttk.Label(frm, text="Y axis").grid(row=0, column=2, sticky="w")
        self.chart_y = ttk.Combobox(frm, state="readonly", width=25)
        self.chart_y.grid(row=1, column=2, sticky="ew", padx=(0,10))

        ttk.Button(frm, text="Plot Line", command=self.plot_chart).grid(row=1,
column=3, sticky="w")

        # canvas holder
        self.chart_area = ttk.Frame(self.tab_chart)
        self.chart_area.pack(fill="both", expand=True, padx=10, pady=(0,10))

    # Actions
    def browse_file(self):
        path = filedialog.askopenfilename(
            title="Select Excel file",
            filetypes=[("Excel files", "*.xlsx *.xls *.xlsm *.xlsb"), ("All files",
"*.*")]
        )
        if path:
            self.file_var.set(path)
            self.populate_sheets()

    def populate_sheets(self):
        path = self.file_var.get()
        if not path:
            return
        try:
            xl = pd.ExcelFile(path, engine="openpyxl")
```

```
            self.sheet_cmb["values"] = xl.sheet_names
            if xl.sheet_names:
                self.sheet_cmb.current(0)
            self.status_var.set(f"Found sheets: {', '.join(xl.sheet_names)}")
        except Exception as e:
            messagebox.showerror("Error", f"Could not read Excel file:\n{e}")

    def load_excel(self):
        path = self.file_var.get()
        sheet = self.sheet_cmb.get()
        if not path or not sheet:
            messagebox.showwarning("Missing", "Please choose a file and sheet.")
            return
        try:
            self.df = pd.read_excel(path, sheet_name=sheet, engine="openpyxl")
            self.current = self.df.copy()
            show_dataframe_in_tree(self.tree_preview, self.df)
            self._refresh_column_pickers()
            self.status_var.set(f"Loaded: {os.path.basename(path)} | Sheet: {sheet} |
Rows: {len(self.df)}")
        except Exception as e:
            messagebox.showerror("Error", f"Failed to load sheet:\n{e}")

    def _refresh_column_pickers(self):
        if self.df is None:
            return
        cols = list(self.df.columns)

        # Summary
        num_cols = [c for c in cols if pd.api.types.is_numeric_dtype(self.df[c])]
        self.summary_num_cols.delete(0, tk.END)
        for c in num_cols:
            self.summary_num_cols.insert(tk.END, c)
        self.summary_group["values"] = [""] + cols
        self.summary_group.set("")

        # Filter
        self.filter_col["values"] = cols
        if cols:
```

```
            self.filter_col.current(0)

        # Pivot
        self.pivot_index["values"] = cols
        self.pivot_columns["values"] = cols
        self.pivot_values["values"] = cols
        if cols:
            self.pivot_index.current(0)
            self.pivot_columns.current(0)
            # Prefer first numeric for values
            cand = next((c for c in cols if pd.api.types.is_numeric_dtype(self.df[c])),
cols[0])
            self.pivot_values.set(cand)

        # Chart
        self.chart_x["values"] = cols
        self.chart_y["values"] = cols
        if cols:
            self.chart_x.current(0)
            self.chart_y.current(0)

    def use_preview_as_current(self):
        if self.df is None:
            return
        self.current = self.df.copy()
        self.status_var.set("Set current result to original data.")

    def compute_summary(self):
        if self.current is None and self.df is None:
            messagebox.showwarning("No data", "Load data first.")
            return
        data = self.current if self.current is not None else self.df

        selected = [self.summary_num_cols.get(i) for i in
self.summary_num_cols.curselection()]
        if not selected:
            messagebox.showwarning("Missing", "Select at least one numeric
column.")
            return
```

```
        aggs = []
        if self.agg_mean.get(): aggs.append("mean")
        if self.agg_std.get():  aggs.append("std")
        if self.agg_min.get():  aggs.append("min")
        if self.agg_max.get():  aggs.append("max")
        if not aggs:
            messagebox.showwarning("Missing", "Pick at least one aggregation.")
            return

        grp = self.summary_group.get().strip()
        try:
            if grp:
                result = data.groupby(grp)[selected].agg(aggs)
            else:
                result = data[selected].agg(aggs).T
            # neat columns
            if isinstance(result.columns, pd.MultiIndex):
                result.columns = ["_".join([str(c) for c in col if c != ""]) for col in
result.columns]
            self.current = result.reset_index()
            show_dataframe_in_tree(self.tree_summary, self.current)
            self.status_var.set("Summary computed. Current result updated.")
        except Exception as e:
            messagebox.showerror("Error", f"Summary failed:\n{e}")

    def apply_filter(self):
        if self.current is None and self.df is None:
            messagebox.showwarning("No data", "Load data first.")
            return
        data = self.current if self.current is not None else self.df

        col = self.filter_col.get()
        op = self.filter_op.get()
        val_raw = self.filter_val.get()

        if not col or not op:
            messagebox.showwarning("Missing", "Choose column and operator.")
            return
```

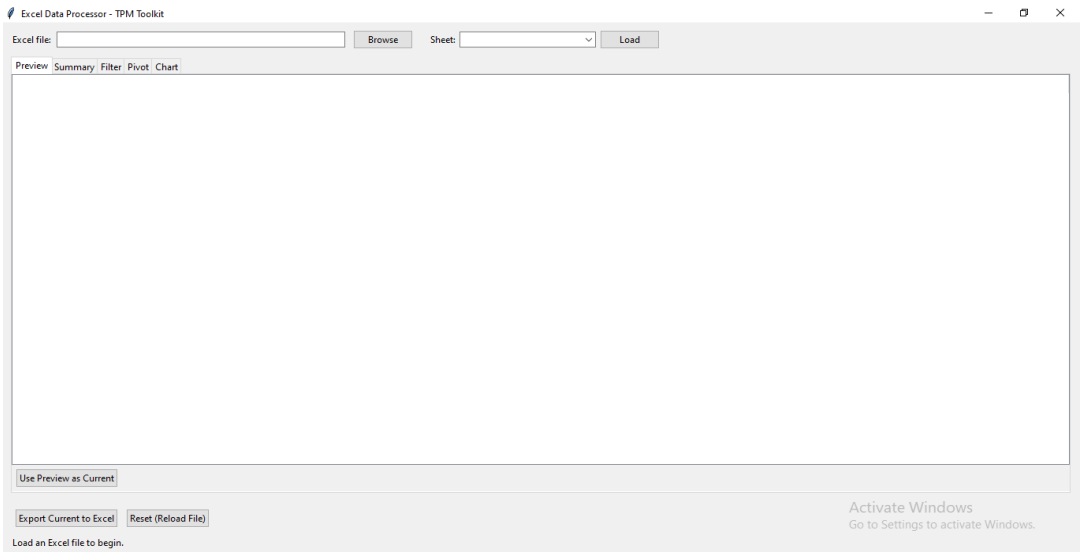
```
        try:
            series = data[col]
            # Try cast numeric if both column and value look numeric
            val = val_raw
            if pd.api.types.is_numeric_dtype(series):
                try:
                    val = float(val_raw)
                except:
                    # allow string compare on numeric col
                    pass

            if op.startswith("=="):
                mask = series == val
            elif op.startswith("!="):
                mask = series != val
            elif op == "contains":
                mask = series.astype(str).str.contains(str(val), na=False, case=False)
            elif op == ">":
                mask = series > val
            elif op == ">=":
                mask = series >= val
            elif op == "<":
                mask = series < val
            elif op == "<=":
                mask = series <= val
            else:
                mask = pd.Series([True]*len(series), index=series.index)

            result = data[mask].copy()
            self.current = result
            show_dataframe_in_tree(self.tree_filter, result)
            self.status_var.set(f"Filter applied. Rows: {len(result)}. Current result
updated.")
        except Exception as e:
            messagebox.showerror("Error", f"Filter failed:\n{e}")

    def build_pivot(self):
        if self.current is None and self.df is None:
```

```
            messagebox.showwarning("No data", "Load data first.")
            return
        data = self.current if self.current is not None else self.df

        idx = self.pivot_index.get().strip()
        cols = self.pivot_columns.get().strip()
        val = self.pivot_values.get().strip()
        agg = self.pivot_agg.get().strip()

        if not (idx and cols and val and agg):
            messagebox.showwarning("Missing", "Pick index, columns, values and
agg.")
            return
        try:
            result = pd.pivot_table(data, index=idx, columns=cols, values=val,
aggfunc=agg)
            result = result.reset_index()
            # Flatten columns if MultiIndex
            if isinstance(result.columns, pd.MultiIndex):
                result.columns = ["_".join([str(c) for c in col if c != ""]) for col in
result.columns]
            self.current = result
            show_dataframe_in_tree(self.tree_pivot, result)
            self.status_var.set("Pivot built. Current result updated.")
        except Exception as e:
            messagebox.showerror("Error", f"Pivot failed:\n{e}")

    def _get_chart_df(self):
        if self.chart_source.get() == "Original data":
            return self.df
        return self.current

    def plot_chart(self):
        df = self._get_chart_df()
        if df is None or df.empty:
            messagebox.showwarning("No data", "Nothing to plot.")
            return

        xcol = self.chart_x.get().strip()
```

```
        ycol = self.chart_y.get().strip()
        if not (xcol and ycol):
            messagebox.showwarning("Missing", "Choose X and Y.")
            return
        if xcol not in df.columns or ycol not in df.columns:
            messagebox.showwarning("Invalid", "Selected columns not in data.")
            return

        # Clear previous canvas
        for w in self.chart_area.winfo_children():
            w.destroy()

        try:
            fig = Figure(figsize=(8,4), dpi=100)
            ax = fig.add_subplot(111)
            ax.plot(df[xcol].values, df[ycol].values)
            ax.set_xlabel(xcol)
            ax.set_ylabel(ycol)
            ax.set_title(f"{ycol} vs {xcol}")

            canvas = FigureCanvasTkAgg(fig, master=self.chart_area)
            canvas.draw()
            canvas.get_tk_widget().pack(fill="both", expand=True)
            self.status_var.set("Chart rendered.")
        except Exception as e:
            messagebox.showerror("Error", f"Plot failed:\n{e}")

    def export_current(self):
        if self.current is None or self.current.empty:
            messagebox.showwarning("Nothing to export", "No current result
available.")
            return
        path = filedialog.asksaveasfilename(
            title="Save Excel",
            defaultextension=".xlsx",
            filetypes=[("Excel files", "*.xlsx")]
        )
        if not path:
            return
```

```
        try:
            with pd.ExcelWriter(path, engine="openpyxl") as writer:
                self.current.to_excel(writer, index=False, sheet_name="Result")
                if self.df is not None:
                    self.df.to_excel(writer, index=False, sheet_name="Original")
            self.status_var.set(f"Exported to: {os.path.basename(path)}")
            messagebox.showinfo("Exported", f"Saved: {path}")
        except Exception as e:
            messagebox.showerror("Error", f"Export failed:\n{e}")

    def reset_current(self):
        if self.df is None:
            return
        self.current = self.df.copy()
        self.status_var.set("Reset current result to original data.")

if __name__ == "__main__":
    app = ExcelProcessorApp()
    app.mainloop()
```

After running the code will open a tkinter window as shown below and use sample Excel files to process data .



Python Development Setup with IntelliJ IDEA

After an Excel file is selected and a sheet is loaded into the application, the user can perform various data processing operations through the tabbed interface. These include :

- Preview
- Summary
- Filter
- Pivot
- Chart

Each function works on either the original or current dataset, which can be transformed and updated through the operations. Below table indicates various operations and purposes.

Operations	Purposes
Preview	Displays the loaded Excel sheet in a scrollable table view
Summary	Grouping and aggregation (mean, std, min, max) across numeric columns
Filter	Applies conditional filtering to the dataset
Pivot	Creates pivot tables for grouped data summarization
Chart	Visualize data as a line chart using Matplotlib