Working with Excel Files in Python

1. Why Work With Excel Files in Python?

Excel files are commonly used for:

- reports
- financial records
- employee data
- sales data
- inventories
- student records
- project tracking
- business dashboards
- exported application data

Doing repetitive Excel work manually can take hours.

Python can automate many of these tasks:

Excel File

Python

Read Data

Clean / Transform

Analyze

Create / Update Workbook

`Save Excel File`

For tabular data, **pandas** provides `read_excel()` and `to_excel()` for reading and writing Excel data. pandas represents the imported table as a `DataFrame`.

For more direct workbook manipulation—such as working with individual cells, formatting, worksheets, formulas, and workbook structure—**openpyxl** is commonly used for modern Excel files.

---

2. Excel Files You Will Encounter

Common Excel extensions include:

```
.xlsx
.xls
.xlsm
.xlsb
```

The most common modern format is:

```
.xlsx
```

A workbook can contain multiple worksheets:

```
report.xlsx
□
□□□ Sales
□□□ Customers
□□□ Products
□□□ Summary
```

Think of:

Workbook = Excel file
Worksheet = individual sheet
Cell = individual location
Row = horizontal record
Column = vertical field

---

3. Choosing the Right Python Tool

There are two major approaches.

pandas

Use pandas when your primary goal is:

- reading tabular data
- filtering
- cleaning
- transforming
- aggregating
- analyzing
- converting data
- creating data reports

Example:

```
import pandas as pd

df = pd.read_excel("sales.xlsx")
```

pandas reads Excel data into a `DataFrame`.

---

openpyxl

Use openpyxl when you need more direct control over the workbook itself.

Examples:

- individual cells
- formulas
- styles
- fonts
- fills
- borders
- worksheet creation
- row/column dimensions
- merged cells
- workbook structure

A useful mental model is:

```
pandas
   
Data-focused

openpyxl
   
Workbook-focused
```

---

4. Installing pandas and openpyxl

For pandas:

```
pip install pandas
```

For openpyxl:

```
pip install openpyxl
```

pandas uses Excel engines to read and write spreadsheet formats; for modern `.xlsx` files, `openpyxl` is a standard engine.

A project that uses both can have:

`requirements.txt`

```
pandas
openpyxl
```

---

5. Reading an Excel File With pandas

Suppose you have:

`sales.xlsx`

Read it:

```
import pandas as pd

df = pd.read_excel("sales.xlsx")

print(df)
```

The result is a DataFrame.

For example:

```
Name      Product      Amount
Hafsa     Course       5000
Asim      Book         2500
Ali       Course       4500
```

pandas automatically interprets the spreadsheet as structured tabular data.

---

6. Inspecting the Data

After reading the workbook:

```
print(df.head())
```

See the shape:

```
print(df.shape)
```

See column names:

```
print(df.columns)
```

See data types:

```
print(df.dtypes)
```

Get a summary:

```
print(df.info())
```

A useful first step is:

```
df.head()
df.shape
df.columns
df.dtypes
```

This tells you what you are working with before you start modifying anything.

---

7. Reading a Specific Sheet

Suppose the workbook contains:

```
Sales
Customers
Products
```

Read `Sales`:

```
df = pd.read_excel(
    "report.xlsx",
    sheet_name="Sales"
)
```

You can also use the sheet position:

```
df = pd.read_excel(
    "report.xlsx",
    sheet_name=0
```

```
)
```

The first sheet has index `0`, the second `1`, and so on.

---

8. Reading Multiple Sheets

You can request several sheets:

```
data = pd.read_excel(
    "report.xlsx",
    sheet_name=["Sales", "Customers"]
)
```

The result is a dictionary-like collection of DataFrames.

```
sales = data["Sales"]
customers = data["Customers"]
```

You can also load all sheets:

```
data = pd.read_excel(
    "report.xlsx",
    sheet_name=None
)
```

Now:

```
for sheet_name, df in data.items():
    print(sheet_name)
    print(df.head())
```

pandas documents `sheet_name=None` as returning a dictionary containing a DataFrame for each sheet.

---

9. Using ExcelFile

When working with multiple worksheets, `ExcelFile` can make the workflow clearer.

```
import pandas as pd

with pd.ExcelFile("report.xlsx") as workbook:

    print(workbook.sheet_names)

    sales = pd.read_excel(
        workbook,
        "Sales"
    )

    customers = pd.read_excel(
        workbook,
        "Customers"
    )
```

`ExcelFile` exposes `sheet_names` and can parse individual sheets.

---

10. Selecting Specific Columns

Suppose the workbook contains:

`Name`

Email
Department
Salary
JoiningDate

But you only need:

Name
Email
Department

Use:

```
df = pd.read_excel(
    "employees.xlsx",
    usecols=[
        "Name",
        "Email",
        "Department"
    ]
)
```

You can also specify Excel column ranges:

```
df = pd.read_excel(
    "employees.xlsx",
    usecols="A:C"
)
```

pandas supports column names, integer positions, and Excel-style ranges through `usecols`.

---

11. Reading Excel Without a Header

Sometimes the spreadsheet has no header row.

Example:

```
Hafsa    25
Asim     28
Ali      23
```

Use:

```
df = pd.read_excel(
    "people.xlsx",
    header=None
)
```

Then pandas uses numeric column labels:

```
0
1
```

You can provide your own names:

```
df = pd.read_excel(
    "people.xlsx",
    header=None,
    names=["Name", "Age"]
)
```

---

12. Skipping Rows

Suppose an Excel file begins with:

```
Company Report
Generated: September 2026

Name    Sales
Hafsa   5000
Asim    7000
```

The actual table begins later.

You can skip rows:

```
df = pd.read_excel(
    "report.xlsx",
    skiprows=2
)
```

pandas supports `skiprows` for ignoring unwanted rows during import.

---

13. Reading Specific Data Types

Sometimes you need explicit types.

```
df = pd.read_excel(
    "employees.xlsx",
    dtype={
        "EmployeeID": str,
        "Name": str
    }
)
```

This can be important for values such as:

`00123`

If treated as a number, it may become:

`123`

while you may actually need:

`00123`

pandas allows explicit `dtype` specifications for imported columns.

---

14. Working With Data After Reading

Once the Excel file becomes a DataFrame, you can use normal pandas operations.

For example:

```
high_sales = df[
    df["Amount"] > 5000
]
```

Sort:

```
df = df.sort_values("Amount")
```

Group:

```
summary = df.groupby(
    "Department"
)["Salary"].sum()
```

Add a column:

```
df["Tax"] = df["Salary"] * 0.10
```

This is where pandas becomes powerful:

```
Excel
 
DataFrame
 
Python data processing
```

---

15. Cleaning Excel Data

Real-world spreadsheets are rarely perfect.

You may find:

```
Hafsa
Hafsa
 hafsa
HAFSA
```

or missing values:

```
Hafsa
None
Asim
```

You can clean values:

```
df["Name"] = df["Name"].str.strip()
```

Handle missing values:

```
df = df.dropna()
```

Or:

```
df["Salary"] = df["Salary"].fillna(0)
```

The important principle is:

```
Read
 
Inspect
 
Clean
 
Validate
 
Process
```

---

16. Writing a DataFrame to Excel

After processing data:

```
df.to_excel(
    "output.xlsx",
    index=False
)
```

The `index=False` prevents the DataFrame's numeric index from becoming an extra Excel column.

pandas provides `DataFrame.to_excel()` for writing DataFrames to Excel.

---

17. Choosing a Sheet Name

```
df.to_excel(
    "output.xlsx",
    sheet_name="Sales Report",
    index=False
)
```

Now the workbook contains:

```
output.xlsx
□□□ Sales Report
```

---

18. Writing Multiple Sheets

Suppose you have:

```
sales = ...
customers = ...
```

You can write both into one workbook:

```
with pd.ExcelWriter("report.xlsx") as writer:

    sales.to_excel(
        writer,
        sheet_name="Sales",
        index=False
```

```
    )

    customers.to_excel(
        writer,
        sheet_name="Customers",
        index=False
    )
```

`ExcelWriter` is designed for writing DataFrames to multiple Excel sheets, and pandas recommends using it as a context manager.

---

19. Creating an Excel Report

A useful workflow is:

```
import pandas as pd

sales = pd.read_excel("sales.xlsx")

sales["Total"] = (
    sales["Quantity"]
    * sales["Price"]
)

summary = (
    sales
    .groupby("Product")["Total"]
    .sum()
    .reset_index()
)

with pd.ExcelWriter("sales_report.xlsx") as writer:
```

```
    sales.to_excel(
        writer,
        sheet_name="Detailed Sales",
        index=False
    )

    summary.to_excel(
        writer,
        sheet_name="Summary",
        index=False
    )
```

The output workbook becomes:

```
sales_report.xlsx
□
□□□ Detailed Sales
□□□ Summary
```

---

20. Working Directly With Excel Using openpyxl

Sometimes pandas is not enough.

Install:

```
pip install openpyxl
```

Import:

```
from openpyxl import load_workbook
```

Open an existing workbook:

```
workbook = load_workbook(
    "report.xlsx"
)
```

Get a worksheet:

```
sheet = workbook["Sales"]
```

Read a cell:

```
value = sheet["A1"].value

print(value)
```

---

21. Creating a Workbook With openpyxl

```
from openpyxl import Workbook

workbook = Workbook()

sheet = workbook.active

sheet["A1"] = "Name"
sheet["B1"] = "Score"

sheet["A2"] = "Hafsa"
sheet["B2"] = 95

workbook.save("students.xlsx")
```

The resulting workbook contains:

```
Name      Score
Hafsa     95
```

---

22. Working With Rows and Columns

You can access:

```
sheet["A1"]
```

or:

```
sheet.cell(
    row=1,
    column=1
)
```

Example:

```
cell = sheet.cell(
    row=2,
    column=1
)

print(cell.value)
```

This is useful when your program calculates cell positions dynamically.

---

23. Iterating Through Rows

```
for row in sheet.iter_rows():
```

```
    for cell in row:
        print(cell.value)
```

You can also retrieve values only:

```
for row in sheet.iter_rows(
    values_only=True
):
    print(row)
```

This produces tuples such as:

```
("Hafsa", 95)
("Asim", 88)
```

---

24. Adding Rows

```
sheet.append(
    ["Hafsa", 95]
)
```

Add another:

```
sheet.append(
    ["Asim", 88]
)
```

This is useful for generating spreadsheets programmatically.

Example:

```
students = [
    ["Hafsa", 95],
    ["Asim", 88],
    ["Ali", 91]
]

for student in students:
    sheet.append(student)
```

---

25. Reading Existing Excel Data With openpyxl

```
from openpyxl import load_workbook

workbook = load_workbook(
    "students.xlsx"
)

sheet = workbook["Sheet"]

for row in sheet.iter_rows(
    values_only=True
):
    print(row)
```

This approach gives you direct access to workbook cells rather than immediately converting the spreadsheet into a DataFrame.

---

26. Creating Worksheets

```
summary = workbook.create_sheet(
    "Summary"
```

)

Now:

```
Workbook
□
□□□ Sheet
□□□ Summary
```

You can write:

```
summary["A1"] = "Total Students"
summary["B1"] = 50
```

---

27. Renaming Worksheets

```
sheet.title = "Students"
```

Now:

```
Workbook
□□□ Students
```

---

28. Deleting Worksheets

```
workbook.remove(sheet)
```

Be careful when deleting sheets from important workbooks.

For production automation:

Load

Validate workbook

Make intended change

Save to a new file

Verify

is safer than immediately overwriting the original.

---

29. Formatting Cells

openpyxl lets you control workbook presentation.

For example:

```
from openpyxl.styles import Font

sheet["A1"].font = Font(
    bold=True
)
```

Now the header is bold.

---

30. Changing Font Properties

```
sheet["A1"].font = Font(
    name="Arial",
```

```
    size=12,
    bold=True
)
```

You can control properties such as:

- font name
- size
- bold
- italic
- underline

---

31. Cell Fills

```
from openpyxl.styles import PatternFill

sheet["A1"].fill = PatternFill(
    fill_type="solid",
    fgColor="D9EAF7"
)
```

This can make report headers easier to identify.

---

32. Borders

```
from openpyxl.styles import Border, Side

thin = Side(style="thin")

sheet["A1"].border = Border(
```

```
    left=thin,
    right=thin,
    top=thin,
    bottom=thin
)
```

Borders can be useful for report tables.

---

33. Alignment

```
from openpyxl.styles import Alignment

sheet["A1"].alignment = Alignment(
    horizontal="center"
)
```

You can also control:

```
Alignment(
    horizontal="center",
    vertical="center"
)
```

---

34. Number Formatting

Suppose:

```
sheet["B2"] = 12345.67
```

You can specify:

```
sheet["B2"].number_format = "#,##0.00"
```

For percentages:

```
sheet["C2"].number_format = "0.00%"
```

For dates:

```
sheet["D2"].number_format = "YYYY-MM-DD"
```

Number formatting changes how values are displayed without necessarily changing the underlying value.

---

35. Column Width

Long text can make a spreadsheet difficult to read.

```
sheet.column_dimensions["A"].width = 25
```

For multiple columns:

```
sheet.column_dimensions["A"].width = 25
sheet.column_dimensions["B"].width = 15
```

---

36. Row Height

```
sheet.row_dimensions[1].height = 25
```

This can be useful for report headers.

---

37. Merging Cells

For a report title:

```
sheet.merge_cells(
    "A1:D1"
)
```

Then:

```
sheet["A1"] = "Monthly Sales Report"
```

The title can span multiple columns.

Use merged cells sparingly in data tables because they can make programmatic processing more complicated.

---

38. Freeze Panes

For large spreadsheets, freeze the header row:

```
sheet.freeze_panes = "A2"
```

Now the first row stays visible while scrolling.

This is particularly useful for:

```
1000+ rows
```

of data.

---

39. Formulas

Excel formulas can be written into cells.

```
sheet["C2"] = "=A2+B2"
```

Another example:

```
sheet["D2"] = "=B2*C2"
```

Python is not necessarily calculating the formula itself.

Instead, it is writing the formula into the workbook.

When Excel opens the workbook, Excel can calculate the formula.

This distinction matters.

```
Python
 
Writes formula
 
Excel
 
Calculates formula
```

---

40. Reading Formula Results

When using openpyxl, loading a workbook normally gives you the formula itself.

```
workbook = load_workbook(
    "report.xlsx"
)

print(
    workbook["Sales"]["D2"].value
)
```

You may see:

```
=B2*C2
```

If you need the cached calculated result instead, you can load with:

```
workbook = load_workbook(
    "report.xlsx",
    data_only=True
)
```

Then:

```
print(
    workbook["Sales"]["D2"].value
)
```

The returned value depends on whether a previously saved workbook contains a cached calculation result.

---

41. Working With Dates

Excel dates can become Python date/datetime values.

With pandas:

```
df = pd.read_excel(
    "employees.xlsx",
    parse_dates=["JoiningDate"]
)
```

With openpyxl:

```
value = sheet["A2"].value

print(value)
```

You may receive a Python date/datetime object depending on the cell.

For reliable processing, always understand how the source workbook represents dates.

---

42. Excel Files and CSV Files Are Different

You already learned CSV files.

CSV:

```
Name,Age,Department
Hafsa,25,CS
Asim,28,IT
```

Excel:

```
Workbook
 □□□ Sheet1
```

□□□ Sheet2
□□□ Formatting
□□□ Formulas
□□□ Data

CSV is fundamentally a text-based table.

Excel is a structured workbook format.

Therefore:

CSV
□
Simple tabular data

Excel
□
Tabular data + workbook features

---

43. pandas vs openpyxl

Task	pandas	openpyxl
Read tabular Excel data	Excellent	Good
Analyze data	Excellent	Limited
Filter rows	Excellent	Manual

Group data	Excellent	Manual
Create DataFrame	Native	No
Individual cell control	Limited	Excellent
Formatting	Some support	Excellent
Formulas	Limited	Direct workbook control
Worksheets	Good	Excellent
Workbook structure	Limited	Excellent
Data transformation	Excellent	Manual

The choice depends on the task.

Use pandas when:

`Data → Process → Analyze → Export`

Use openpyxl when:

`Workbook → Inspect/Edit/Format → Save`

---

44. Combining pandas and openpyxl

You do not have to choose only one.

A powerful workflow is:

Excel

pandas

Clean + Analyze

ExcelWriter

openpyxl

Apply formatting

Final workbook

For example:

```
import pandas as pd
from openpyxl import load_workbook

df = pd.read_excel("sales.xlsx")

df["Total"] = df["Quantity"] * df["Price"]

df.to_excel(
    "report.xlsx",
    index=False
)
```

```
workbook = load_workbook("report.xlsx")

sheet = workbook.active

sheet["A1"].font = Font(
    bold=True
)

workbook.save("report.xlsx")
```

This separates:

```
Data processing
```

from:

```
Workbook presentation
```

---

45. Updating an Existing Workbook

Sometimes you don't want to create a new file.

You may need to update an existing workbook.

With pandas:

```
with pd.ExcelWriter(
    "report.xlsx",
    mode="a",
    engine="openpyxl"
```

```
) as writer:

    df.to_excel(
        writer,
        sheet_name="New Data",
        index=False
    )
```

For more precise workbook modifications, use openpyxl.

Always understand whether you are:

```
creating
overwriting
appending
updating
```

because these operations have different consequences.

---

46. Safe Excel Automation

Suppose:

```
original.xlsx
```

is an important file.

Instead of:

```
# overwrite immediately
```

prefer:

```
original.xlsx
      
Read
      
Process
      
output.xlsx
      
Validate
```

This gives you a recoverable original.

For business-critical spreadsheets, backups and validation are especially important.

---

47. Validating Excel Data

Suppose a sales spreadsheet requires:

```
Name
Product
Quantity
Price
```

Before processing:

```
required_columns = {
    "Name",
    "Product",
    "Quantity",
```

```
    "Price"
}

missing = required_columns - set(df.columns)

if missing:
    raise ValueError(
        f"Missing columns: {missing}"
    )
```

This prevents your automation from silently processing the wrong spreadsheet.

---

48. Handling Missing Values

Example:

```
if df["Price"].isna().any():
    print("Some prices are missing.")
```

Or:

```
df["Price"] = df["Price"].fillna(0)
```

But don't automatically replace missing data without understanding what it means.

For example:

```
Missing salary
```

does not necessarily mean:

`Salary = 0`

Data validation should reflect the business rules.

---

49. Excel Automation Pipeline

A robust Excel automation process can look like:

```
Input Workbook
    ↓
Validate File
    ↓
Inspect Sheets
    ↓
Read Data
    ↓
Validate Columns
    ↓
Clean Data
    ↓
Transform Data
    ↓
Generate Summary
    ↓
Write Workbook
    ↓
Format Workbook
    ↓
Validate Output
    ↓
Save
```

This is much safer than:

Open

Change

Save

---

50. Automating a Monthly Report

Imagine a company receives:

January.xlsx
February.xlsx
March.xlsx

You can automate:

Find files

Read each workbook

Combine data

Calculate totals

Create summary

Generate final Excel report

Conceptually:

```
import pandas as pd
from pathlib import Path

files = Path("monthly").glob("*.xlsx")

frames = []

for file in files:
    df = pd.read_excel(file)
    frames.append(df)

combined = pd.concat(
    frames,
    ignore_index=True
)

combined.to_excel(
    "annual_report.xlsx",
    index=False
)
```

This combines Excel handling with the file automation concepts you already learned.

---

51. Mini Project: Excel Sales Report Generator

Build a program that receives:

```
sales.xlsx
```

containing:

```
Date
```

Salesperson
Product
Quantity
Price

Your program should:

1. Read the workbook.
2. Validate required columns.
3. Remove invalid rows.
4. Calculate total sales.
5. Group sales by salesperson.
6. Group sales by product.
7. Create a summary sheet.
8. Create a detailed sheet.
9. Format headers.
10. Freeze the header row.
11. Adjust column widths.
12. Save the final report.

Output:

sales_report.xlsx
□
□□□ Detailed Sales
□□□ By Salesperson
□□□ By Product

This is a strong real-world Python automation project.

---

52. 5 Minute Challenge

Create:

`students.xlsx`

with:

```
Name
Math
English
Computer
```

Write Python that:

1. Reads the workbook.
2. Calculates each student's average.
3. Adds an `Average` column.
4. Creates a `Result` column.
5. Saves `students_result.xlsx`.

Example rule:

```
Average >= 50  Pass
Average < 50   Fail
```

Then make the header bold using openpyxl.

---

53. Exercises

Exercise 1

Read an Excel workbook with pandas.

Exercise 2

Print all worksheet names.

Exercise 3

Read only one worksheet.

Exercise 4

Read multiple worksheets.

Exercise 5

Select specific columns.

Exercise 6

Filter rows based on a value.

Exercise 7

Calculate a new column.

Exercise 8

Export a DataFrame to Excel.

Exercise 9

Create a workbook with multiple sheets.

Exercise 10

Create a workbook using openpyxl.

Exercise 11

Add rows programmatically.

Exercise 12

Format a header.

Exercise 13

Freeze the first row.

Exercise 14

Add formulas.

Exercise 15

Build an automated Excel report.

---

54. Mini Quiz

1. What does `pd.read_excel()` return?

A. A string
B. A DataFrame
C. A list only

D. A workbook object

Answer: B

2. Which library is particularly useful for data analysis?

A. pandas
B. smtplib
C. pathlib
D. logging

Answer: A

3. Which library provides direct cell and workbook manipulation?

A. openpyxl
B. requests
C. json
D. csv

Answer: A

4. What does `index=False` do in `to_excel()`?

A. Deletes the first row
B. Prevents the DataFrame index from being written as a column
C. Deletes the index from the DataFrame
D. Prevents Excel from opening

Answer: B

5. What does `sheet_name` specify?

A. File extension
B. Worksheet to read or write
C. Python module
D. Cell formatting

Answer: B

6. What does `sheet_name=None` do with `read_excel()`?

A. Reads no sheets
B. Reads all sheets into a dictionary of DataFrames
C. Deletes all sheets
D. Creates a new workbook

Answer: B

---

55. Interview Questions

Beginner

1. How do you read an Excel file in Python?
2. What is a DataFrame?
3. How do you write a DataFrame to Excel?
4. How do you select a particular worksheet?
5. What is `openpyxl`?
6. What is the difference between `.xlsx` and `.csv`?
7. How do you create a new Excel workbook?

8. How do you read a specific cell using openpyxl?

Intermediate

9. When would you use pandas instead of openpyxl?
10. How do you read multiple Excel sheets?
11. How do you write multiple DataFrames into one workbook?
12. How do you add a worksheet with openpyxl?
13. How do you modify cell formatting?
14. How do you freeze rows in Excel?
15. How do you write formulas?
16. How do you validate required Excel columns?
17. How would you process hundreds of Excel files?
18. How would you prevent accidental overwriting of an original workbook?

Advanced

19. How would you design an automated Excel reporting pipeline?
20. How would you combine pandas data processing with openpyxl formatting?
21. How would you handle malformed or inconsistent spreadsheets?
22. How would you validate an output workbook?
23. How would you process large Excel files efficiently?
24. How would you preserve formulas and formatting while modifying workbook data?
25. How would you make Excel automation reliable enough for scheduled production jobs?

---

56. Excel Files Cheat Sheet

Read Excel

```
import pandas as pd

df = pd.read_excel(
```

```
    "data.xlsx"
)
```

Read specific sheet

```
df = pd.read_excel(
    "data.xlsx",
    sheet_name="Sales"
)
```

Read all sheets

```
data = pd.read_excel(
    "data.xlsx",
    sheet_name=None
)
```

Select columns

```
df = pd.read_excel(
    "data.xlsx",
    usecols=["Name", "Salary"]
)
```

Write Excel

```
df.to_excel(
    "output.xlsx",
    index=False
)
```

Multiple sheets

```
with pd.ExcelWriter(
    "report.xlsx"
) as writer:
```

```
    df.to_excel(
        writer,
        sheet_name="Data",
        index=False
    )
```

Create workbook

```
from openpyxl import Workbook

workbook = Workbook()
sheet = workbook.active

workbook.save("data.xlsx")
```

Open workbook

```
from openpyxl import load_workbook

workbook = load_workbook(
    "data.xlsx"
)
```

Select sheet

```
sheet = workbook["Sales"]
```

Read cell

```
value = sheet["A1"].value
```

Write cell

```
sheet["A1"] = "Hello"
```

Add row

```
sheet.append(
    ["Hafsa", 95]
)
```

Bold

```
from openpyxl.styles import Font

sheet["A1"].font = Font(
    bold=True
)
```

Freeze header

```
sheet.freeze_panes = "A2"
```

Save

```
workbook.save(
    "output.xlsx"
)
```

---

57. The Excel Automation Mental Model

Remember:

```
UNDERSTAND
What does the spreadsheet represent?
        
INSPECT
What sheets, columns and data types exist?
```

VALIDATE
Is the workbook structured as expected?

READ
Load the data with pandas or openpyxl.

PROCESS
Clean, filter, transform and calculate.

WRITE
Create or update the workbook.

FORMAT
Make the report readable when necessary.

VERIFY
Check that the output contains the expected data.

The goal is not:

> Make Python edit an Excel file.

The goal is:

> Turn repetitive spreadsheet work into a reliable, repeatable data workflow.

---

58. Key Takeaways

- Excel workbooks can be automated extensively with Python.
- pandas is particularly useful when Excel data needs to be analyzed or transformed.
- `read_excel()` loads Excel data into DataFrames.
- `to_excel()` writes DataFrames back to Excel.
- `ExcelWriter` is useful for generating workbooks containing multiple sheets.
- openpyxl gives more direct control over workbook structure and individual cells.
- A workbook can contain multiple worksheets.
- You can create, rename, delete, and modify worksheets.
- openpyxl can modify fonts, fills, borders, alignment, number formats, dimensions, and other workbook features.
- Excel formulas can be written programmatically.
- `data_only=True` can be useful when reading cached formula results.
- Real-world Excel automation should validate input data before processing it.
- Important source workbooks should not be overwritten blindly.
- pandas and openpyxl can be combined when you need both data processing and workbook formatting.
- Good Excel automation follows a pipeline: **read → validate → process → write → format → verify**.

Website:
https://dev-roast-app.vercel.app

More resources:
https://dev-roast-app.vercel.app/resources

Visit haas.dev for more resources and guides.