Modern Data Analytics in Excel

Transform, Model, and Analyze Data in Spread-sheets

George Mount

Beijing • Boston • Farnham • Sebastopol • Tokyo O'REILLY®

CHAPTER 12
Python with Excel

Thus far in the book we've been looking at all the things Excel can do. Power Query and Power Pivot aren't the only big new things. There is growing interoperability with Python. Let's take a look at that in this chapter.

Reader prerequisites

Although this chapter can be completed without prior knowledge of Python, familiarity with concepts such as lists, indexing, and the `pandas` and `seaborn` packages will greatly enhance your understanding.

If you'd like to learn more about Python before reading, I recommend starting with my book, *Advancing into Analytics: From Excel to Python and R* (O'Reilly, 2021), for a basic introduction to Python for Excel users. To delve even deeper into the subject, check out Felix Zumstein's *Python for Excel: A Modern Environment for Automation and Data Analysis* (O'Reilly, 2021).

This chapter includes a hands-on Python coding demonstration. To fully benefit from it, I encourage you to follow along on your own computer. All you need is a completely free version of Python. I recommend downloading the Anaconda distribution, which is available at [anaconda.com](*https://www.anaconda.com/*).

The Role of Python in Modern Excel

Between Power Query and Power Pivot alone, "modern" Excel boasts so many features that it's impossible to learn them all. Add to that Office Scripts, Power BI, `LAMBDA()` and more and it's easy to get overwhelmed.

Prospective learners often find Python, an Excel-adjacent tool, intimidating. Many Excel users believe it should be the last tool on their learning list since it's not a Microsoft product and requires acquiring a new language proficiency.

Python may not be the best choice for every Excel user, but it's worth serious consideration for those looking to build complex automations, version-controlled projects and other advanced development products. Let's explore the role of Python in modern analytics and its relationship with modern Excel.

A growing stack requires glue

When I started as an analyst, my choices started and ended with Excel. All data, reporting, dashboards, everything was under that one green-and-white roof.

Fast forward a few years and there's now Power BI, Office Scripts, Jupyter Notebooks and more. This broadening of the analyst's tech stack is in line with wider trends in tech: a move away from one single, monolithic application to a loosely connected suite of specialized products.

To make this architecture work, a "conductor" or "glue" application is needed. Whether it's reading a dataset in from one source and visualizing it another, or deploying a machine learning model from the cloud to an end-user's dashboard, Python is a great choice for that glue. It's one of the rare languages used for simple amateur-written scripts to enterprise-level applications alike, and it can work smoothly with a variety of operating systems and other programming languages.

Microsoft has acknowledged and celebrated Python's role as a "glue" language, and it's already available to use for a variety of purposes in Azure, Power BI, SQL Server and more. Although Python is not currently officially supported for Excel, it still offers significant advantages in Excel-related tasks, as you will discover in this chapter. Additionally, Python has gained widespread adoption among developers and organizations, making it a language with a thriving community and extensive resources.

Network effects mean faster development time

"Everybody's doing it" isn't usually a good reason to engage in something, but in the case of programming languages it may hold merit

Network effects, the concept that the value of something increases with its user base, apply to programming languages. As more programmers join, code sharing expands, providing a larger codebase for usage and further development, creating a virtuous cycle.

Python's versatility as a neutral "glue" language has led to its adoption in various professions, including database management, web development, data analytics, and more. This implies that regardless of the direction your Excel project takes or the tools you require, there's a high likelihood of finding collaborators who "speak" Python.

For example, you might develop an inventory tracker or other application using Excel. The program gets too unwieldy for a workbook, or it becomes so popular that your organization wants it to become a standalone web program. The turnaround time for this project will be much faster if the existing code is already in Python.

It's unfortunate that data analysts and tech professionals often use different tools, leading to slow and cumbersome collaboration. However, by embracing a shared language such as Python, the network effects amplify, resulting in accelerated development and reduced time requirements.

Bring modern development to Excel

The job title of "software developer" has gained popularity in recent years, often being seen as desirable as the role of a "data scientist." However, the effective methods and best practices developed by software developers have not been widely adopted by BI or VBA developers. This situation raises concerns at the organizational level, as having two technology professionals labeled as "developers" but following distinct methodologies can lead to complications.

Python enables modern Excel developers to implement best practices, such as:

Unit testing

Many programming languages provide automated unit tests to ensure code functions correctly. However, Excel lacks this feature. Although there are some workaround tools available, Python is a strong choice for unit testing due to its network effects and abundance of packages. Automated unit testing enhances reliability and reduces the likelihood of errors, which is especially valuable for Excel workbooks accessed by users with diverse technical backgrounds.

Version control

Unlike most modern development tools, Excel lacks a version control system. This system tracks changes in a repository, enabling users to view contributions, revert to previous versions, and more. If you've ever struggled to differentiate between multiple workbooks like *budget-model-final.xlsx* and *budget-model-FINAL-final.xlsx*, you can appreciate the usefulness of version control.

Although Excel offers limited version control features like viewing version history in OneDrive and using the Spreadsheet Compare add-in, it falls short compared to the extensive features available when transitioning code production to Python.

Package development and distribution

If you're seeking a more immediate reason to learn Python for your daily analysis tasks, then let me introduce you to one word: packages.

While I enjoy developing my own tolls, I believe in leveraging existing solutions when they fit my needs. Python's powerful features for building and sharing packages, particularly through the Python Package Index, unlock a world of tools that are challenging to replicate with Excel add-ins or VBA modules.

Whether you want to gather data from an application programming interface (API), analyze images, or simply generate descriptive statistics, the availability of Python packages justifies the investment in learning Python. Some of these packages are even designed to seamlessly integrate with Excel.

Python and the future of Excel

In an AI-driven world, learning to code may appear to be the wrong choice for staying relevant. Ironically, as data continues to grow in various formats, including AI-powered ones, the importance of coding becomes more relevant than ever.

Python's integration with Excel is becoming increasingly expansive, aligning with the modern analytics stack and the incorporation of AI-driven features within Excel.

Using Python and Excel together with `pandas` and `open pyxl`

With Python's role in modern Excel in mind, let's explore how the two can work together. Two key packages to facilitate this integration are `pandas` and `openpyxl`. Let's consider the two in turn.

Why `pandas` for Excel?

If you're working with any kind of tabular data in Python, you won't get far without `pandas`\`. This package lets you, among other operations:

- Sort and filter rows
- Add, remove and transform columns
- Aggregate and reshape a table
- Merge or append multiple tables

This is Python's equivalent to Power Query, enabling you to create reusable data cleaning and transformation workflows. And just like Power Query, `pandas` can effortlessly import data from diverse sources, including Excel, and even export the results of your analysis back to Excel.

The limitations of working with pandas for Excel

That said, pandas has limited features for deeply interacting with Excel workbooks. For example, it cannot help with the following tasks:

- Advanced formatting options for cells, such as applying specific styles or conditional formatting
- Support for executing Excel macros or VBA code within workbooks
- Direct access to Excel-specific features like data validation, charts, pivot tables, or formulas
- The ability to manipulate worksheets, such as renaming, adding, or deleting sheets
- Fine-grained control over workbook properties, such as password protection or worksheet visibility
- Handling of Excel-specific file formats like .xlsb or .xlsm
- Integration with Excel add-ins or plugins

Fortunately, several packages exist to provide these more advanced Python/Excel features, most notably openpyxl.

What `openpyxl` contributes

`openpyxl` (pronounced *open pie Excel*) is a Python package providing functionality for working with Excel files, specifically the newer *.xlsx* file format. It allows users to read, write, and modify Excel spreadsheets programmatically. openpyxl integrates smoothly with pandas, allowing users to clean data using pandas and add additional functionality to the workbook using openpyxl.

Specifically, openpyxl can help with the following tasks where `pandas` cannot:

- Advanced formatting options for cells, such as applying specific styles or conditional formatting
- The ability to manipulate worksheets, such as renaming, adding, or deleting sheets
- Fine-grained control over workbook properties, such as password protection or worksheet visibility
- Working with named ranges and tables
- Adding images, shapes, and charts to Excel files
- Handling print settings, page layout, and page breaks

- Working with formulas and formula-related functionality in Excel

Although `openpyxl` has limitations and cannot cover every use case, such as some mentioned earlier, it remains the most powerful and accessible Python package for automating Excel tasks.

How to use `openpyxl` with `pandas`

Let's take a typical use case for automating a routine Excel business report where an analyst needs to generate monthly sales reports from multiple Excel worksheets. The analyst might read the data from each worksheet into `pandas` DataFrames, then continue to use `pandas` to clean and analyze the data. Finally, `openpyxl` is used to generate a consolidated report in a new Excel workbook, which contains conditional formatting, charts, and more. The analyst could then use other Python tools to automate the distribution of the report.

For these and other tasks, the basic workflow for using pandas with openpyxl is like so:

1. Read the data: Use pandas to extract data from a variety of sources into tabular DataFrames
2. Clean and analyze the data: Use pandas to clean and manipulate the data, perform calculations, apply filters, handle missing values, and derive relevant insights.
3. Generate the report: Use openpyxl to create a new Excel workbook or select an existing one. Populate the workbook with the consolidated data, applying conditional formatting, creating charts, and incorporating any required visual elements.
4. Save the report: Save the updated Excel workbook using openpyxl, specifying the desired filename and location.
5. Distribute and automate the report: Send the generated report to the intended recipients through email, file sharing platforms, or any preferred method.

Other Python packages for Excel

Powerful as it is for Excel tasks, especially when combined with `pandas`, `openpyxl` has limitations. Thankfully, other packages are available to handle specific use cases. Some other packages to be aware of:

- `xlsxwriter`: Similar to `openpyxl`, `xlsxwriter` can be used to write data, formatting, and charts to Excel files in the *.xlsx* format. This package is optimized for performance, particularly when working with large datasets. It also offers more advanced cell formatting options compared to openpyxl. That said, as the name implies, xlsxwriter can only handle writing data to Excel, while openpyxl can both read and write.

- xlwings: This package enables the automation of Excel tasks, including interacting with Excel workbooks, running VBA macros, and accessing Excel's COM (Component Object Model) API on Windows. It provides complete two-way communication between Excel and Python in a way that openpyxl does not. On the other hand, this package requires a much more complex development environment, with many features only available on Windows.
- pyxll: This is a paid library that enables users to write Excel add-ins using Python. Instead of automating Excel workbooks, pyxll allows developers to build standalone applications for data science, financial trading, and other purposes.

Many other Python packages exist for Excel-related tasks, each with unique strengths and weaknesses. Considering Python's vast and active user community, the possibility of even more Excel-focused packages emerging is likely.

Demonstration of Excel automation with pandas and openpyxl

Let's stop discussing and start building. In this section we'll automate production of a small report from Python using pandas, openpyxl and more.

First, we will use pandas to perform complex data cleaning and analysis tasks that are difficult to achieve in Excel. After that, we will create an overview worksheet consisting of summary statistics and two charts, one from native Excel and one from Python. Finally, we'll load the entire dataset to a new worksheet and format the results.

A completed version of this script is available as ch_12.ipynb in the ch_12 folder of the book's companion repository. If you're not sure how to open, navigate or interact with this file, check out Part 3 of *Advancing into Analytics: From Excel to Python and R* (O'Reilly, 2021) as a primer to Python and Jupyter Notebooks.

Let's import the relevant modules and dataset to get started:

```
In [1]: import pandas as pd
        import seaborn as sns
        from openpyxl import Workbook
        from openpyxl.chart import BarChart, Reference
        from openpyxl.drawing.image import Image
        from openpyxl.utils.dataframe import dataframe_to_rows
        from openpyxl.utils import get_column_letter
        from openpyxl.utils.dataframe import dataframe_to_rows
        from openpyxl.worksheet.table import Table, TableStyleInfo
        from openpyxl.styles import PatternFill
```

The pandas library can import data from various formats, such as Excel workbooks using the read_excel() function. Let's read the contestants.xlsx file as contest ants.

```
In [2]: contestants = pd.read_excel('data/contestants.xlsx')
```

Cleaning up the data in pandas

A pandas DataFrame may have thousands of rows or more, so printing them all is impractical. However, it is important to visually inspect the data — a benefit Excel users are familiar with. To get a glimpse of the data and ensure it meets our expectations, we can use the head() method to display the first five rows.

```
In [3]: contestants.head()

Out[3]:

 EMAIL  COHORT  PRE     POST    AGE     SEX     EDUCATION       SATISFACTION    STUDY_HOURS
 0      smehaffey0@creativecommons.org  4       485     494     32.0    Male    Bachelor's236.6
 1      dbateman1@hao12@.com    4       462     458     33.0    Female  Bachelor's822.4
 2      bbenham2@xrea.com       3       477     483     NaN     Female  Bachelor's119.8
 3      mwison@@g.co    2       480     488     31.0    Female  Bachelor's      1033.1
 4      jagostini4@wordpress.org        1       495     494     38.0    Female  NaN932.5
```

Based on this data preview, we identified there are a few issues that need to be addressed. First, it appears that some of the emails contain an invalid format. We also have some values in the AGE and EDUCATION columns called NaN which don't seem to belong. We can address these and other issues in the dataset in ways that would be difficult or impossible to do with Excel's features.

Working with the metadata

A good data analysis and transformation program should be equally proficient in handling both data and metadata. In this regard, pandas stands out as a particularly suitable tool.

Currently, our DataFrame has column names in uppercase. To make typing column names easier, I prefer using lowercase names. Fortunately, in pandas, we can accomplish this with a single line of code:

```
In [4]: contestants.columns = contestants.columns.str.lower()
        contestants.head()

Out[4]:

email   cohort  pre     post    age     sex     education       satisfaction    study_hours
0       smehaffey0@creativecommons.org  4       485     494     32.0    Male    Bachelor's236.6
```

```
1       dbateman1@hao12@.com    4       462     458     33.0    Female  Bachelor's822.4
2       bbenham2@xrea.com       3       477     483     NaN     Female  Bachelor's119.8
3       mwison@@g.co    2       480     488     31.0    Female  Bachelor's      1033.1
4       jagostini4@wordpress.org        1       495     494     38.0    Female  NaN932.5
```

Pattern matching/regular expressions

The `email` column of this DataFrame contains email addresses for each contest participant. Our task is to remove any rows from this column that have invalid email addresses.

Text pattern matching like this is accomplished using a set of tools known as regular expressions. While Power Query does offer basic text manipulation capabilities, such as case conversions, it lacks the ability to search for specific patterns of text, a feature available in Python.

Regular expressions can be challenging to create and validate, but there are online resources available to assist with that. Here is the regular expression we will be using:

```
In [5]: # Define a regular expression pattern for valid email addresses
        email_pattern = r'^[a-z0-9]+[\._]?[a-z0-9]+[@]\w+[.]\w{2,3}$'
```

Next, we can use the `str.contains()` method to keep only the records that match the pattern.

```
In [6]: full_emails = contestants[contestants['email'].str.contains(email_pattern)]
```

To confirm how many rows have been filtered out, we can compare the `shape` attribute of the two DataFrames:

```
In [7]: # Dimensions of original DataFrame
        contestants.shape

Out[7]: (100, 9)

In [8]: # Dimensions of DataFrame with valid emails ONLY
        full_emails.shape

Out[8]: (82, 9)
```

Analyzing missing values

The `info()` method offers a comprehensive overview of the DataFrame's dimensions and additional properties:

```
In [9]: full_emails.info()

    <class 'pandas.core.frame.DataFrame'>
    Int64Index: 82 entries, 0 to 99
    Data columns (total 9 columns):
    #   Column        Non-Null Count  Dtype
    ---  ------        --------------  -----
    0   email         82 non-null     object
```

```
1   cohort        82 non-null     int64
2   pre           82 non-null     int64
3   post          82 non-null     int64
4   age           81 non-null     float64
5   sex           82 non-null     object
6   education     81 non-null     object
7   satisfaction  82 non-null     int64
8   study_hours   82 non-null     float64
dtypes: float64(2), int64(4), object(3)
memory usage: 6.4+ KB
```

In Python and other programming languages, `null` refers to a missing or undefined value. In pandas DataFrames, this is typically denoted as `NaN`, which stands for "Not a Number."

While basic Excel lacks an exact equivalent to `null`, Power Query does provide this value, which greatly aids in data management and inspection. However, it can be difficult to programatically work with these missing values in Power Query. pandas makes this easier.

For example I might want to see what columns have the most percentage of missing values. I can do this easily with `pandas`:

```
In [10]: full_emails.isnull().mean().sort_values(ascending=False)

Out[10]:

        age             0.012195
        education       0.012195
        email           0.000000
        cohort          0.000000
        pre             0.000000
        post            0.000000
        sex             0.000000
        satisfaction    0.000000
        study_hours     0.000000
        dtype: float64
```

Because there are so few missing values, we will simply drop any row that has a missing observation in any column:

```
In [11]: complete_cases = full_emails.dropna()
```

To confirm that all missing observations have been cleared from the DataFrame, we can use the `info()` method again:

```
In [12]: complete_cases.info()

        <class 'pandas.core.frame.DataFrame'>
        Int64Index: 80 entries, 0 to 99
        Data columns (total 9 columns):
        #   Column        Non-Null Count  Dtype
```

```
---  ------         --------------  -----
0   email         80 non-null     object
1   cohort        80 non-null     int64
2   pre           80 non-null     int64
3   post          80 non-null     int64
4   age           80 non-null     float64
5   sex           80 non-null     object
6   education     80 non-null     object
7   satisfaction  80 non-null     int64
8   study_hours   80 non-null     float64
dtypes: float64(2), int64(4), object(3)
memory usage: 6.2+ KB
```

Creating a percentile

Using `pandas`, we'll create a percentile rank for the `post` column and confirm its validity by running descriptive statistics with `describe()`:

```
In [13]: complete_cases['post_pct'] = complete_cases['post'].rank(pct=True)
         complete_cases['post_pct'].describe()

Out[13]:

        count    80.000000
        mean      0.506250
        std       0.290375
        min       0.012500
        25%       0.264062
        50%       0.506250
        75%       0.756250
        max       1.000000
        Name: post_pct, dtype: float64
```

Creating a percentile column in Excel is straightforward, but validating results is easier in `pandas`. It offers statistical functions, methods for handling missing values, and more.

Our dataset has been cleaned and transformed using `pandas`:

```
In [14]: complete_cases.describe()

Out[14]:

            cohort      pre     post    age     satisfaction    study_hours     post_pct
        count   80.000000       80.000000       80.000000       80.000000       80.00000080.000000
        mean    2.475000        480.550000      480.987500      30.162500       5.23750028.316250
        std     1.147093        20.752856       23.181995       6.042976        2.8694985.228245
        min     1.000000        409.000000      398.000000      20.000000       1.00000014.500000
        25%     1.000000        470.000000      466.500000      25.000000       3.00000025.950000
        50%     2.500000        484.000000      483.000000      30.500000       5.00000028.050000
        75%     3.250000        494.000000      497.000000      35.000000       7.00000032.225000
        max     4.000000        521.000000      540.000000      40.000000       10.00000040.900000
```

Now let's use openpyxl to create a styled summary report.

Summarize findings with openpyxl

Creating a summary worksheet

To get started building an Excel workbook with openpyxl, we'll declare variables representing workbook and worksheet objects:

```
In [14]: # Create a new workbook and select the worksheet
         wb = Workbook()

         # Assign the active worksheet to ws
         ws = wb.active
```

From there, we can populate any cell of the active sheet using its alphanumeric reference. I am going to insert and label the average pre and post scores in cells A1:B2:

```
In [16]: ws['A1'] = "Average pre score"
         ws['B1'] = round(complete_cases['pre'].mean(), 2)  # Round output to two decimals
         ws['A2'] = "Average post score"
         ws['B2'] = round(complete_cases['post'].mean(), 2)
```

Inserting data into the workbook this way is just a raw data dump; it does not affect the appearance of the data in Excel. Given my experience formatting data, I suspect the labels in column A will require more width. I'll adjust that now via the width property of the worksheet:

```
In [17]: ws.column_dimensions['A'].width = 16
```

Later in this chapter, we'll cover achieving AutoFit-like adjustments for column widths. But for now, let's shift our focus to adding charts to this summary.

Inserting charts

Option A: Create a native Excel plot

Excel's data visualization features are popular because they are easy to use and effective. Let's explore how to create native Excel charts from Python using openpyxl.

To begin, we need to specify the type of Excel chart we want to create and identify the location of the data for the chart within the worksheet:

```
In [18]: # Create a bar chart object
         chart = BarChart()

         # Define the data range
         data = Reference(ws, min_col=2, min_row=1, max_col=2, max_row=2)
```

Next, we'll add this data source to the chart and label the chart's title and axes:

```
In [19]: # Add data to the chart
         chart.add_data(data)
```

```
          # Set chart title, axis labels
          chart.title = "Score Comparison"
          chart.x_axis.title = "Score Type"
          chart.y_axis.title = "Score Value"
```

Let's further customize this chart. We'll set category labels to reflect the data in the first column and also eliminate the chart legend.

```
In [20]: # Set category names
         categories = Reference(ws, min_col=1, min_row=1, max_row=2)
         chart.set_categories(categories)

         # Remove the legend
         chart.legend = None
```

With the chart fully defined and styled, it's time to insert it into the worksheet.

```
In [21]: # Add the chart to a specific location on the worksheet
         ws.add_chart(chart, "D1")
```

Option B: Insert a Python image

Python offers advantages in data visualization compared to Excel, as it provides more diverse visualization options and allows for easier customization of plots. For example, Excel lacks a built-in solution for analyzing relationships between multiple variables simultaneously. However, the `seaborn` data visualization package offers the `pairplot()` function, which provides a quick and convenient way to explore such relationships.

The following block visualizes these relationships across the selected variables of contestants:

```
In [22]: sns.pairplot(contestants[['pre', 'post', 'age', 'study_hours']])
```

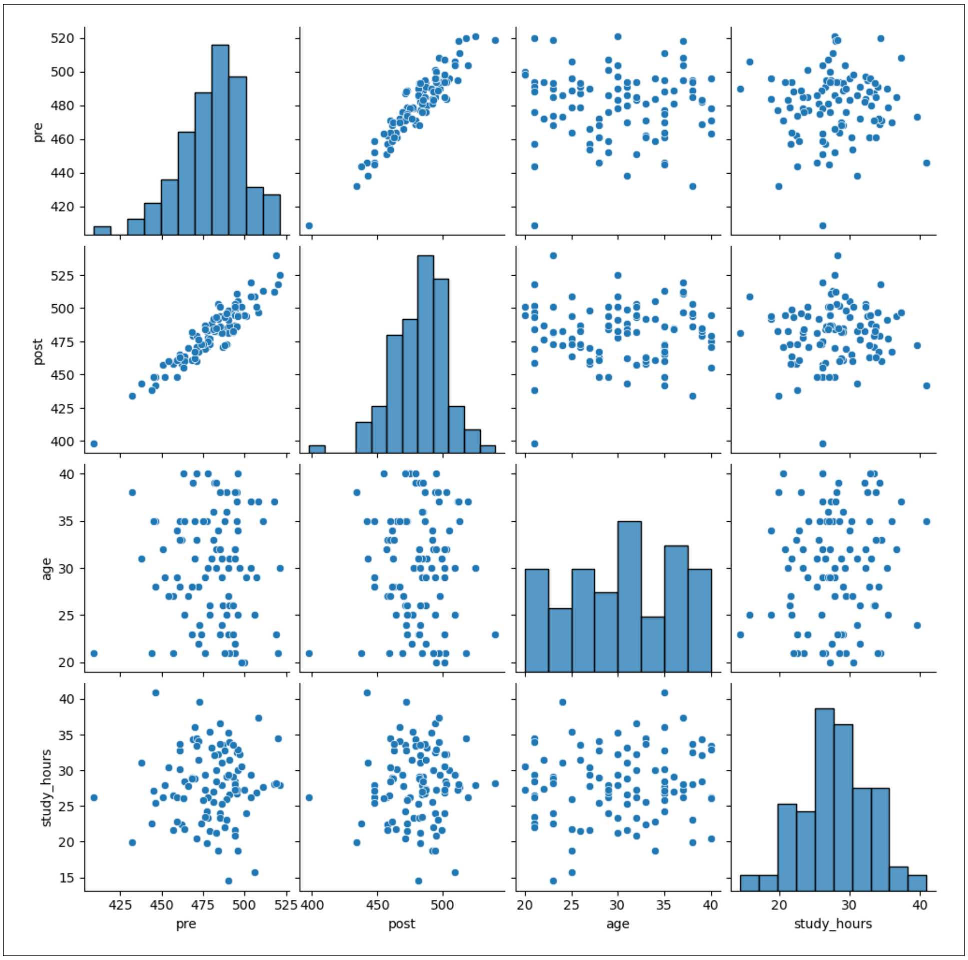

Figure 12-1. Pairplot

Not only does Python include a number of chart types that are difficult to build in Excel, they are easy to customize as well. For example, I'd like to see this visualization broken down by sex, which can be done by passing it to the hue parameter. I'm going to save the results of this plot as sns_plot so I can refer to it later.

```
In [23]: sns_plot = sns.pairplot(contestants[['pre', 'post', 'age', 'study_hours', 'sex']],
             hue='sex')
```

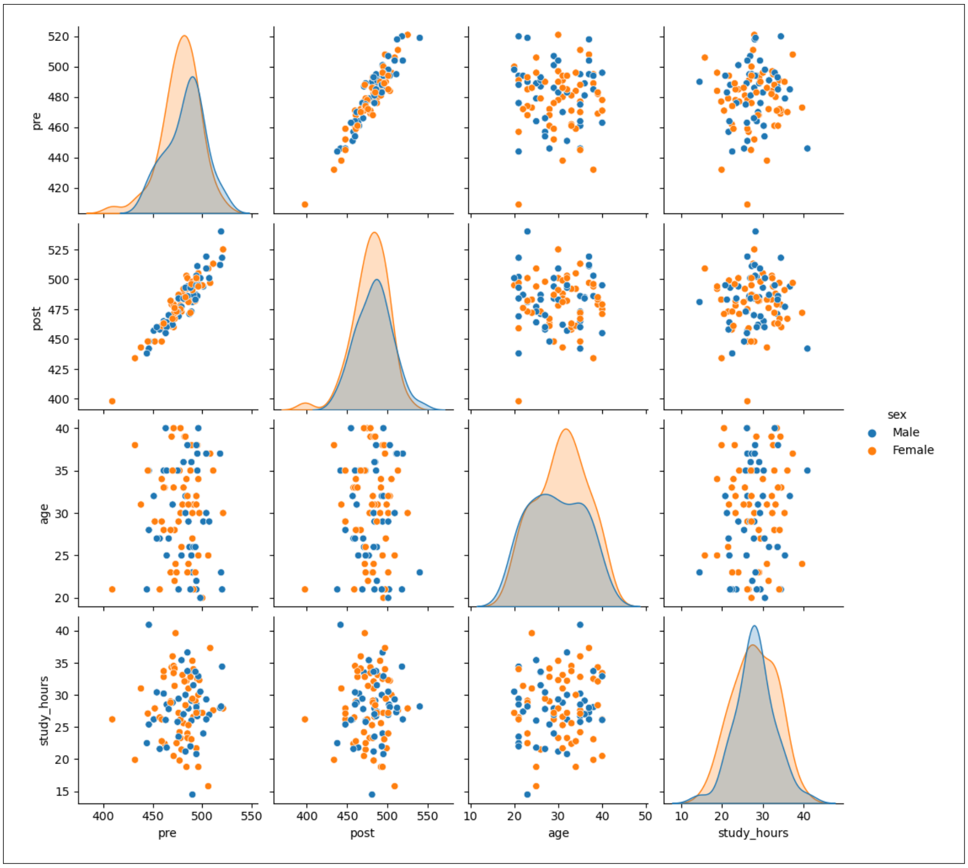

Figure 12-2. Pairplot by sex

Next, let's place a static image of this pairplot into the workbook. This requires first saving the image to disk, then specifying where to place it into the workbook:

```
In [21]: # Save pairplot to disk as an image
         sns_plot.savefig('pairplot.png')

         # Load saved image into the worksheet
         image = Image('pairplot.png')
         ws.add_image(image, 'A20')
```

Excel versus Python charts

A summary of the pros and cons of these two methods follows in Table 12-1:

Table 12-1. Pros and cons of Python versus Excel charts

	Pros	Cons
Building a native Excel plot	- Plot will update with changes in Excel data User can interact with and customize plot Plot can integrate with other Excel features like formulas and PivotTables	- Limited number of Excel plot types exist Some complex calculations or statistical functions may be easier to build in Python
Inserting an image of a Python plot	- Access to several powerful plotting libraries such as matplotlib and seaborn Plot is easily audited and reproduced through the source code	- The plot is a static image and lacks interactivity Updating or refreshing the chart from Excel is not possible

The choice between methods depends on different factors, such data refresh needs and the availability of specific chart types in Excel. That said, the flexibility and range of options available in itself highlight Python's powerful capabilities for working with Excel.

Adding a styled data source

Now that our summary worksheet has been created, we will create a second, styled worksheet consisting of the `complete_cases` DataFrame. The first step is to define this new worksheet:

```
In [25]: ws2 = wb.create_sheet(title='data')
```

Next, we'll loop over each row of `complete_cases` and insert them into the worksheet:

```
In [26]: for row in dataframe_to_row(complete_cases, index=False, header=True):
    ws2.append(row)
```

Inserting the DataFrame into the worksheet is a start, but the resulting data may be challenging for users to read and manipulate. Let's make a few improvements.

Formatting percentages

By default, the `post_pct` column will be formatted in Excel as decimals instead of more readable percentages. To address this, we need to specify the location of this column in the worksheet and re-format it.

I will use the `get_loc()` method to find the index position of this column in the DataFrame, adding 1 to the results to account for Excel's 1-based indexing. The `get_column_letter()` function will convert this index number into Excel's alphabetical column referencing.

```
In [27]: post_pct_loc = complete_cases.columns.     get_loc('post_pct') + 1
         post_pct_col = get_column_letter(post_pct_loc)
         post_pct_col
```

```
Out[27]: 'J'
```

With the proper column identified, I will apply the desired number formatting to each row:

```
In [28]: number_format = '0.0%'

         for cell in ws2[post_pct_col]:
             cell.number_format = number_format
```

Converting to a table

As discussed in the Preface of this book, tables hold a number of benefits for data storage and analysis. We can convert this dataset into an Excel table with the following code:

```
In [29]: # Specify desired table formatting
         style = TableStyleInfo(name='TableStyleMedium9', showRowStripes=True)

         # Name and identify range of table
         table = Table(displayName='contestants',
                    ref='A1:' + get_column_letter(ws2.max_column) + str(ws2.max_row))

         # Apply styling and insert in worksheet
         table.tableStyleInfo = style
         ws2.add_table(table)
```

Applying conditional formatting

To enhance readability for end users, we can apply conditional formatting to the worksheet. The following code will apply a green background fill to participants above the 90th percentile and a yellow background fill to participants above the 70th percentile:

```
In [30]: # Define conditional formatting style
         green_fill = PatternFill(start_color="B9E8A2", end_color="B9E8A2", fill_type="solid")
         yellow_fill = PatternFill(start_color="FFF9D4", end_color="FFF9D4", fill_type="solid")

         # Loop through data table and conditonally apply formatting
         for row in ws2.iter_rows(min_row=2, min_col=1, max_col=len(complete_cases.columns)):
             post_pct = row[post_pct_loc - 1].value # Convert index to 0-based indexing
             if post_pct > .9:
                 for cell in row:
                     cell.fill = green_fill
             elif post_pct > .7:
                 for cell in row:
                     cell.fill = yellow_fill
```

Auto-fitting column widths

Although `openpyxl` lacks an AutoFit feature to automatically resize worksheet columns, we can achieve a similar outcome with the following code. It finds the widest row in each column of the worksheet, then adds enough padding to adjust the column width accordingly:

```
In [31]: for column in ws2.columns:
         max_length = 0
         column_letter = column[0].column_letter
         for cell in column:
             try:
                if len(str(cell.value)) > max_length:
                max_length = len(cell.value)
            except:
                pass
         adjusted_width = (max_length + 2) * 1.2
         ws2.column_dimensions[column_letter].width = adjusted_width
```

After completing the workbook, we can save the results to `ch12-output.xlsx`.

```
In [32]: wb.save('ch12-output.xlsx')
```

Conclusion

This chapter explored Python's role in modern Excel, highlighting its versatility as a "glue" language for development and its ability to enhance Excel's capabilities. Through a hands-on demo, it demonstrated how Python automates Excel, adding features difficult or impossible to achieve within the spreadsheet program alone. As Microsoft further integrates Python into its data analytics stack, the Python and Excel landscape will evolve. However, this chapter provides a solid framework for effectively using Python and Excel together, maximizing their potential.

Exercises

For this chapter, create a concise summary report of the `websites.xlsx` file located in the `datasets` folder of the book's companion repository. Use the provided `ch_12_starter_script.ipynb` file as a starting point for the project. Fill in the missing sections of the Jupyter notebook to achieve the desired solution, which can be found in the `exercise_solutions` folder of the repository. Refer to the chapter's examples for guidance in writing the code accurately, and feel free to incorporate additional automated features into your work.

To be completed later