When introducing newcomers to Python in Excel, I typically recommend that they have some basic Python knowledge beforehand. This helps reduce the mental effort required to grasp the Python in Excel environment. However, using Python within Excel also assumes a solid understanding of Excel itself.

Like many technological advancements, embedding Python directly into Excel leverages an impressive and robust set of existing features. This post aims to explore and explain these features in a bit more depth. You can follow along using the exercise file provided below:

Download the exercise file here

Power Query

To fully leverage Python in Excel, it’s necessary to have a basic understanding of Power Query. Unfortunately, Python in Excel operates within a closed system, meaning you can’t use Python to pull in external data sources via flat files, APIs, web scraping, and so forth.

That task still falls to Power Query, which is highly capable in its own right. If you’re new to Power Query and want to get familiar with its basic use cases and interface, I recommend checking out my book, Modern Data Analytics in Excel:

You can explore further details about this connection on Microsoft’s support page. For now, let’s dive into a simple example together, where we’ll pull a table of country populations from Wikipedia.

Start by opening a new worksheet in the demo workbook. Then, navigate to the Data tab on the ribbon and select Get Data > From Other Sources > From Web:

Get data from web Power Query

Next, go ahead and paste in the link from Wikipedia as the URL and click OK.

Power Query will analyze the page’s structure and recommend entities to import. I’d like to start with the first HTML table. Click on it to review it, then click Transform Data at the bottom-right to apply a few modifications to the dataset.

Power Query transform data from web

Once we’re in this view, we can perform various data cleaning tasks through a point-and-click interface that generates syntax automatically (at this stage, diving into Power Query with a comprehensive guide like my book is highly recommended!). For instance, I can easily delete the “World” row from the dataset and swiftly rename columns:

Power Query

Every single step we’re doing here gets logged in Applied Steps, so it’s all reproducible, which is music to the ears of coders like Pythonistas jumping into this system. Power Query is definitely a great tool, and I’m looking forward to a day when, like in Power BI, you can just inject Python scripts as one of the steps in a Power Query workflow… that would be the best of both worlds.

Applied steps menu

For now, though, let’s close out this query connection. Go up to the ribbon and hit Close > Close & Load in the upper left. You’ll see a bunch of load options, but I’m gonna pick “Only Create Connection” (no need to dump the data into the Data Model) and click OK.

Close & load Excel

When loading query data in Power Query, the output menu offers options like loading directly into Excel as a table or PivotTable, which displays the final result in the worksheet, unlike the blank worksheet we’re currently seeing. So, why opt for Connection Only at all?

This choice makes sense when you need to preprocess or transform data and keep it on standby for tasks like linking to other queries, establishing relationships, or crafting a data model, all without flooding the worksheet with raw data. It’s particularly useful for managing large datasets or intermediate steps, conserving memory and maintaining a lean workbook while preserving flexibility for later analysis.

To make things really interesting, though, you can still tap into that “Connection Only” data in the workbook using Python code! Before we dive into that, head to the Queries & Connections pane on the right, right-click your query, and rename it to something code-friendly like “world\_pop” for easier handling.

Rename query PQ

Let’s move forward by pulling the Power Query connection data into Python. To do this, use the =PY() function in a cell to switch to Python Mode, then enter this formula to import the data:

wiki\_df = xl("world\_pop")

Press Ctrl + Enter to run the code and import the data.

Import wiki df into Excel

You should now notice a DataFrame displayed in the cell where you imported the data, marked with a [PY] symbol beside it, indicating you’re in Python object view within the worksheet.

While you could switch to Excel Value mode using Ctrl + Alt + Shift + M, staying in Object Mode offers some cool advantages. It keeps things streamlined because we don’t need to display every single value in the dataset, saving both space and processing power. Luckily, there are still user-friendly ways to explore this dataset preview, and that’s where linked data types come into play, giving us a neat way to learn more about this data.

DataFrame of wiki df

Linked data types

Linked data types in Excel let you link cells to external sources, like stock markets or geographic databases, transforming them into vibrant, dynamic data sources. You can learn more about Excel’s native linked data types in this post:

What’s really cool about Python in Excel is how it taps into this linked data type feature to unlock deeper insights into datasets. For instance, clicking the [PY] button in our cell brings up a tooltip-style popup that previews the dataset. You can drag it to expand:

Resize linked data type

Alternatively, you can add a dataset preview right into the worksheet. Just click the table-like icon next to the cell, select the first Field, arrayPreview, and you’re set. You’ll see other attributes available too, mostly tied to the cell’s object type: fitting, since Python is an object-oriented programming language where such details matter.

Linked data type add field

You’ll now see a preview of the data added straight into the worksheet, with the interesting notation A1.arrayPreview to tell us it’s an attribute of a linked data type. We still see some of the data, not all of it, so it’s still a pretty compact view. (This is also leveraging dynamic arrays, which we’ll discuss in more detail shortly.).

Array preview dynamic array

Pulling data straight from Power Query is definitely an efficient way to handle your data. And because Power Query doesn’t have Excel’s usual million-row (give or take) cell limit, you could theoretically manage massive datasets, funnel them through Power Query into Python, and dodge Excel’s row cap.

The main drawback to doing so, in my opinion, is that you’d lose Advanced Analysis in Copilot for AI-powered help with Python coding—though, with datasets so large, Copilot might struggle anyway. I’m hoping for a future where Copilot ties into the whole Excel pipeline and works directly with Power Query data too, but for now, sticking with an Excel table is the best way to use Copilot, so let’s shift from this web data to the much smaller dataset in the nyc-pop worksheet of your workbook.

Excel tables and table references

Even if your dataset is small or already organized in a table, it’s still worth understanding the fundamentals of how this works. This is another topic we can’t fully explore here, so I’d again recommend checking out my book Modern Data Analytics in Excel for a deeper dive. That said, one key benefit of formatting your data as a table is that it becomes a more dependable dynamic data source, thanks to its structured references.

Let’s take a look at this in action with the small nyc table located in the nyc-pop worksheet of the workbook:

Create nyc df Python in Excel

Here, I’ve used PY() function to import our dataset into Python within Excel. You’ll see that with the xl() function, instead of linking to a query, we’re connecting directly to the nyc table using its specific table notation. Since this dataset is so small, I went ahead and switched it from Python object mode to Excel values mode using Ctrl + Alt + Shift + M.

The "#All" in the notation indicates that we’re referencing the entire table—both the data body and the headers (and potentially, at some risk, the footers too). We could adjust this structured reference to explicitly exclude footers, but that gets a bit more advanced, so for now, we’ll stick with this automated setup.

Now that the dataset is loaded into Python in Excel as a DataFrame, I can start applying Pandas operations to it! For example, let’s quickly add a column to calculate population density:

nyc\_df = xl("nyc[#All]", headers=True)

nyc\_df['Density'] = nyc\_df['Population']/nyc\_df['Land area']

nyc\_df

NYC df table

If you’re familiar with Excel tables, you’ll notice this notation feels quite similar to how columns are handled in Excel tables. This is another reason mastering Excel tables is so valuable—even if you’re new to coding, it’s excellent mental preparation!

One of the key advantages of importing this data into Python in Excel as a table, rather than using a static reference, is that it remains tied to the table name, not a fixed cell location. This means if the data’s shape, size, or position changes, the reference adapts automatically. So, if by some wild chance a sixth borough were added to NYC 🗽, this setup would update on its own. It even accounts for the blank row that appeared automatically after I tabbed out of the Test row:

Dynamic arrays and the spill operator

Now that we’re working with the actual Excel values generated by our Python in Excel calculation, let’s explore a couple of Excel features that shape this experience. First up is the dynamic array. You might have noticed that when clicking in and out of our Python in Excel output, the entire range highlights together and seems to pop off the worksheet a big.

This is a sign we’re dealing with a dynamic array—a single formula in one cell that returns a resizable set of outputs across multiple cells. Dynamic arrays come as a package deal: if even one cell obstructs their output range, the whole thing fails to render, and you’ll see a #SPILL error like this:

That’s useful to understand, and it means you should definitely give dynamic arrays plenty of space to avoid issues if their output size changes… they need room to grow!

But things get really exciting when you consider how you can dynamically reference that spill range using the spill operator. For instance, you could calculate the number of rows in the resulting dataset with the ROWS() function (you might want to subtract one to exclude the header), or sum a specific column by combining the SUM() and CHOOSECOLS() functions. This opens the door to all sorts of dynamic calculations and references based on our Python in Excel output.

Spill range dynamic array

Here’s where worlds truly collide: you can leverage Python for tasks that are tricky in Excel, push the results back into Excel, and then use Excel for things that are harder in Python. Even something as simple as formatting the data to make it more readable, which can actually be a bit of a pain in Python, becomes effortless. It’s a powerful mashup of capabilities.

For a more in-depth introduction to dynamic arrays and the spill operator in Excel, you can explore the following two posts and, naturally, my book Modern Data Analytics in Excel.

Images stored in cells

Finally, let’s explore how to use images in cells that plots can render in Python within Excel. For more details on this feature, you can refer to Microsoft Support. Return to the cell we’ve been working with in the nyc-pop worksheet, which contains the NYC dataset, and insert the following code to create a barplot comparing population density across boroughs:

nyc\_df = xl("nyc[#All]", headers=True)

nyc\_df['Density'] = nyc\_df['Population']/nyc\_df['Land area']

sns.barplot(x='Borough', y='Density', data=nyc\_df)

plt.ylabel('Population density per square mile')

plt.title('Population density by borough')

Population density by borough

You should now see a bar plot in your worksheet (ensure you’re in Excel values mode if it’s not visible), though it might appear very small since Python plots are, by default, sized to fit within a single cell.

While storing data directly in a cell has clear benefits—such as being neatly anchored for easier handling and programmatic use (like treating images as input values for functions like VLOOKUP())—the drawback is that these plots can be extremely tiny unless you resize your grid, which may disrupt your workbook’s proportions.

To address this, you can create a reference to the image by clicking on the small image-with-link icon that appears when you select the source code cell:

Expand plot Python in Excel

From this point, you can resize the image or position it wherever you prefer, as it directly references the data in that specific cell (meaning you must not delete the source data cell!).

Conclusion

That wraps up our exploration of Excel features that Python in Excel utilizes! I hope you can see how we’re truly building on the foundation of giants, and that mastering Python in Excel requires a solid understanding of both Python and Excel—a skillset that can set you apart and, in my opinion, will grow even more valuable as AI continues to transform the roles of data analysts and spreadsheet developers.

What questions do you have about these powerful Excel features that enhance Python in Excel? Which ones are you most eager to experiment with? Or, if you’ve already tried them, what use cases have you found beneficial? Share your thoughts in the comments.