This class is for Microsoft Excel users who wish to gain the advantages of a structured programming language to leverage their analytical capabilities.

Excel is an awesome tool for “quick and dirty” calculations and for production of pleasing reports, graphics, and tables. Anyone can use learn to use it quickly and work products easily integrate into other work – especially in a MS Office environment.

Excel’s ease of use comes at a cost, however. Large spreadsheets become unwieldy because of their sheer size. Every manual operation introduces an opportunity for errors, many of which are introduced silently and without obvious warning signals. Spreadsheet manipulations carry no effective audit trail so mistakes can become buried beneath mountains of complexity. Not only are errors difficult to detect, but they can become nearly impossible to unwind once discovered – and the process of unwinding one set of errors can introduce another.

This course will build on your mastery of spreadsheet-based analysis and show how to build Python-based analysis that’s auditable, replicable, and extensible in ways that Excel cannot accomplish.

At the same time, you’ll lose nothing. Python can read and write intact spreadsheets so upstream data producers and downstream clients need not change their workflow. Python’s operations are “plug-compatible” with Excel’s most commonly-used features. Libraries that support testing, statistical analysis, database interactivity, and visualization can be used to extend what’s possible with Excel – and do so without having to switch platforms mid-stream in your analysis.

This course is offered in one-day, two-day, and three-day versions. All assume a solid facility with Excel and basic spreadsheet analysis.

The two-day course is aimed at Excel users with some programming experience in any language. Day 1 provides an introduction to the Python language, version control, testing, and effective use of an interactive development environment. Day 2 provides a crosswalk between already-familiar Excel capabilities and Python’s approach to replicating / extending them. At the end of the course, students will be ready to transition to Python for most analysis and be able to seamlessly integrate Python-produced analytical product with Excel-produced work.

The three-day course is intended for students with facility in Excel, but with no programming experience. Day 1 introduces topics like objects, data types, control loops, program development strategy and debugging. The next two days dive into Python’s analytic features and replication of Excel-like analysis.

The one-day course is aimed at users who have solid experience with Python and mastery of Excel. It immediately dives into Python’s abilities to replicate and extend advanced Excel features. The one-day course is not for the faint-of-heart or for technical new-comers. But upon completion of the course adequately-prepared students with depart with the confidence to leave Excel behind as they tackle their next complex analytical projects.

Customized experiences are also available upon request. We can develop and deliver courses using your group’s present data streams and analytical requirements. Students can work begin with already-familiar information in its raw state, develop Python-based analysis using appropriate tools and libraries, and replicate their current workflow on a more advanced platform.

These courses can include content on statistical techniques, advanced topics like image and signal processing, scientific visualization, database development, integration of your C/C++ libraries into your Python environment, and workflow management for complex projects.

# Basic Python-Based Spreadsheet Operations

## Reading data

## Operations on a column of data

## Creating new columns

## Writing a new spreadsheet

# Spreadsheet Functions

## Using data characteristics – AVERAGE(), MIN(), MAX() , etc.

## Logical statements - IF(), AND, OR

## Lookups – VLOOKUP(), HLOOKUP()

## Sorting data on single and multiple columns

## Create and apply your own spreadsheet functions

# Essential Differences Between Python and Excel

## Excel containers (Worksheet, Workbook, Cell, Range, etc.)

## Python containers (DataFrame, Series, Index, etc.)

## Location of mathematical, statistical, and visualization tools

# Navigating Data

## Excel-like row and column indices

## Upgrading to named indices

## Selecting ranges of rows and columns based on the index

## Finding ranges rows and columns based on the value

## Applying functions to selected ‘ranges’

## Adding rows and columns

# Workflow Topics

## Separation of process flow, utility operations, and parameters

## Creation and application of your own library of worksheet functions

## Testing, validation, and encapsulation

## Create and manage an audit trail

## Using Python to automate inspection of differences in data files

# Data Screening and Cleaning

## Handle missing data = #NULL! , #NA!, #VALUE!, etc.

## Converting data types

## Managing text fields

## Identifying and fixing outliers

## On-the-fly descriptive statistics and visualization

# Combining data from multiple sources

## Options for acquiring data (text, databases, tabbed spreadsheets, etc.)

## Managing indices and duplicate data

## Joining and merging data tables

# Output options and strategies

## Charts in Excel versus Python

## Spreadsheet formatting in Excel versus Python

## Creation and management of multi-tabbed spreadsheets

## Alternative input and output options - serialization, text formats, etc.

# Working with times and dates

## Create datetime indices

## Stretching or compressing time-related data columns

## Business quarters, weekends, holidays

## Elements of time – days, minutes, hours, time zones and time differences

# Optional/Extended Topics

## Pivot tables and hierarchal indices

## Data reduction in Python with (factor analysis, PCA, etc.)

## Classification analysis in Python (LDA, QDA, k-means, etc.)

## Linear regression analysis in Python

## Managing precision in computation and display

## Fuzzy text processing – Excel versus Python

## Image processing and characterization

## Capturing and displaying analysis with Notebooks

## “Introspecting” data and objects with Python tools

## Using Python’s advanced visualization tools

## Creating and managing your Python environment(s)

## Creating and managing your project environment(s)

## Encapsulation at a high level (Docker, AWS, VirtalBox, etc.)

|  |
| --- |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |

There are several options for creating output in Excel format. Among these:

* Use pandas’ to\_excel() method (it’s built into DataFrame and Series objects);
* Write data directly into an existing spreadsheet using an Excel utility like openpyxl;
* Create a spreadsheet from scratch.

Which you choose depends on your workflow, requirements and, frankly, how you want to spend your time. Pandas’ to\_excel() method is easy to use and fairly flexible. This being said, it produces really ugly output.

You can create a spreadsheet from scratch and gain access to fairly granular details about layout, cell formatting. Some of the more refined aesthetic details e.g., coordinated color schemes, are not available from within the Python applications. However, the most commonly-used features e.g., creation of ranges, pivot tables, charts and formatting options like font, display precision, and column widths can be accessed with code.

Access to this detail is a two-edged sword. Yes, you can gain close control over all the details. But, then again, you have to get into a lot of arcane stuff that’s likely not well-aligned with your real goals.

If you want to access most of Excel’s aesthetic wonders without having to delve into the details, there are a couple reasonable “middle ground” strategies.

* Write an Excel macro in VBA to collect data to be displayed from different sources (databases, other spreadsheets, etc.) and ingest them into an already preened and beautiful spreadsheet. This requires a simple Excel macro. But this is a Python class so we’ll discuss another option.
* Write a Python script to move data from some sort of Python object e.g., a DataFrame into a place where an already-developed Excel object can access it.

Either way, you get the option of using Excel’s GUI magic to develop the aesthetic elements of the presentation media and the analytical power of Python to figure out what to display.

Pandas’ to\_excel() Method

The to\_excel() routine is grafted onto all the pandas data objects. All you need to do is specify the name of the file and a crude Excel spreadsheet “just happens”. Here is a simple example:

def create\_spreadsheet():

"sports\_compressed.py"

data = [['baseball', 180, 111], ['wrestling', 30, 300.123],

['gymnastics', 1, 120], ['soccer', 90, 1000.000]]

cols = ['sport', 'duration', 'fans']

#Create the spreadsheet. File name must end with '.xlsx'

pd.DataFrame(data=data, columns=cols).to\_excel("simple.xlsx")

#Create the spreadsheet. File name must end with '.xlsx'

pd.DataFrame(data=data, columns=cols).to\_excel("simple.xlsx")

The sports\_compressed.py script creates this Excel file:

|  |  |  |  |
| --- | --- | --- | --- |
|  | **sport** | **duration** | **fans** |
| **0** | baseball | 180 | 111 |
| **1** | wrestling | 30 | 300.123 |
| **2** | gymnastics | 1 | 120 |
| **3** | soccer | 90 | 1000 |

You’ll note that it’s definitely serviceable – it’s got all the right information, an appropriate number of precision is used. The row and column indices are preserved. But you wouldn’t want to show it to a client. There are a few things you can do using parameters to the function call to help out.

Here’s an alternative formulation of the command, this time working on a named DataFrame:

df = pd.DataFrame(data=data, columns=cols)

df.to\_excel("simple\_1.xlsx", columns = ['fans', 'duration'],

index=False, float\_format = '%.1f',

header=True)

… and the resulting spreadsheet.

|  |  |
| --- | --- |
| **fans** | **duration** |
| 111 | 180 |
| 300.1 | 30 |
| 120 | 1 |
| 1000 | 90 |

A few things are worth noting:

* You can format the precision of floating point numbers using the “olde tyme” format strings[[1]](#footnote-1). This particular example asks for one decimal point of precision. Be a little careful because this controls the actual value of the number sent to the spreadsheet, not just the display precision.
* If you specify which columns to print out, you provide them as a list-like object[[2]](#footnote-2). This lets you leave some out, switch up the order, or even repeat columns.
* You can get rid of the row index, but if you want to retain the column titles you need to do so specifically with the header= parameter.
* The extension you choose for the file name (‘.xls’ versus ‘.xlsx’, typically) cues the system’s choice of the package a.k.a. ‘engine’, used to write the Excel file. You can override it if necessary. If you forget the extension, you’ll get a very strange message about “no suitable engine” – that’s what it’s referring to.

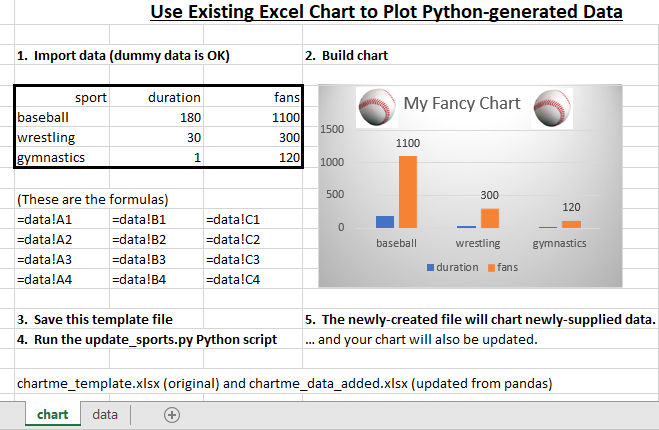
For your reference, here’s an annotated list of the keyword parameters:

|  |  |  |
| --- | --- | --- |
| **Parameter** | **Default** | **Notes** |
| excel\_writer |  | Uses openpyxl library |
| sheet\_name | 'Sheet1' |  |
| na\_rep |  | Substitutes this for NaN |
| float\_format | None | (olde tyme) format string |
| columns | use all | which columns? |
| header | True | colum headings |
| index | True | row index |
| index\_label | None | index column heading |
| startrow | 0 | zero-based index of starting point |
| startcol | 0 |  |
| engine | None | force a writer (otherwise pick on extension) |
| merge\_cells | True | merge columns to display multiple indices? |
| encoding | None | None uses system default or script's shebang |
| inf\_rep | 'inf' | how infinity is displayed (Excel doesn't support) |
| freeze\_panes | None | a (row, column) tuple to 'freeze' headings |

Populating Existing Excel Objects

As mentioned earlier, one way to generate aestetically-pleasing Excel spreadsheets is to prepare a template that has all the necessary elements then ship in data from a Python script. Here is an example from a two-tab spreadsheet – one tab has a Chart object and the other has the data it represents.

Here’s a screen capture from the ‘chart’ tab of chartme.xlsx. It should be self-explanatory, so please have a careful look and consider the points that follow.



The block of data in the bold box shown here is linked to the ‘data’ tab. The Excel SERIES objects shown within the chart are linked to the data in the bold box . From the chart’s perspective, the data could live anywhere. The fact that they’re local is irrelevant, as is just how the local data found their way into the spreadsheet.

To build this chart template, we created an empty spreadsheet with ‘chart’ and ‘data’ tabs. We populated the ‘data’ tab with some dummy data then linked the dummy data to some cells in the chart tab. [[3]](#footnote-3)

The chart reflects whatever aesthetic elements are desired – nothing in the Python script cares[[4]](#footnote-4). The Python script will simply push new data into the ‘data’ tab and allow the chart machinery to do its magic.

The Python code required to add data to the spreadsheet actually uses the same library, openpyxl, that the pandas DataFrame object wraps up in its to\_excel() method. By accessing it directly, we have access to the entire API and can work more intimately with the workbook object[[5]](#footnote-5).

A complete script is available as ‘update\_sports.py’, but the essence of it in the insert\_data\_into\_template() function:

def insert\_data\_into\_template(df, template='', tab\_name='', new\_book\_name=''):

"""Inserts the contents of a DataFrame object into a tab of an existing

spreadsheet and saves the result."""

#Uses pandas' default Excel engine openpyxl directly.

template\_book = load\_workbook(template)

template\_data\_wksh = template\_book[tab\_name]

Here, we use the load\_workbook() method of the openpyxl library to create a Python version of the Excel workbook object. The workbook is a dictionary-like object that provides access to workbook, range, and other internal objects. We create an object to represent the ‘data’ tab.

#We can write the template a cell at a time.

#... the first line will be the columns of the DataFrame

for col, col\_heading in enumerate(df.columns, 2):

#Returns a Cell object, but we don't need it

template\_data\_wksh.cell(row=1, column=col, value = col\_heading)

As is the case with Excel, the business end of the worksheet object is the cell. That’s where the values (data) live and that’s where much of the formatting happens. Here, we’re just concerned in ensuring that the cells take on the values contained in the Python DataFrame. The last line above creates a cell object and provides it a value from the DataFrame’s column header.

A couple of notes here:

* We could have provided a name to the cell object just created and accessed it later, but there’s no need.
* The Excel objects are all “1-based” while Python’s arrays are all “0-based”. This can get confusing when using indices against both. Consider using enumerate() as it’s applied in this example to save much gnashing of teeth[[6]](#footnote-6).

#Write the contents of the DataFrame (note the spreadsheet is "1-based")

rows, cols = df.shape

for out\_row, df\_row in enumerate(range(rows), 2):

this\_row = df.iloc[df\_row, :]

for out\_col, df\_col in enumerate(range(cols), 1):

cell\_obj = template\_data\_wksh.cell(row=out\_row, column=out\_col,

value = this\_row[df\_col])

#This closes the openpyxl workbook object and saves it as a file.

template\_book.save(new\_book\_name)

The script update\_sports.py performs these functions:

* Opens the chart template spreadsheet file
* Updates information in its ‘data’ tab.
* Saves it under a new name.

Et voila, you have a completely atomic Excel chart and its data. We’re admittedly giving up some efficiency here – especially when compared to working with a chart server, on demand production of visualization products, etc. We’re also squandering storage for (mostly) redundant visual elements.

But, it’s easy. It creates a permanent archive of the analytical products. And it completely isolates the technical and aesthetic aspects of your deliverable analytical products.

1. It’s a little-known fact, but you can be the old style strings in the regular string format() expressions. [↑](#footnote-ref-1)
2. You’ll run into the term “list like object” a lot. It generally means “an ordered collection of stuff that Python knows how to iterate over”. A tuple always works – as might a pandas Index, Series, columns, or the like. [↑](#footnote-ref-2)
3. The latter step is mostly for demonstration purposes – so you can see the data that’s plotted here in the same screen shot as the chart itself. It’s not otherwise necessary. [↑](#footnote-ref-3)
4. The Python Excel reader/write objects don’t handle every stylistic nuance, but understand most of them. [↑](#footnote-ref-4)
5. The openpyxl object model is similar to that employed by MS Excel. The top level workbook objects serve as containers for worksheet objects which, in turn, contain cell objects. Workbook objects host named ranges and workbook-centric properties. Worksheets can hold pivot tables, charts, etc. We only scratch the surface here. [↑](#footnote-ref-5)
6. enumerate() can be applied to any iterable object. It returns an iterator that produces an (index\_value, content) tuple for each bit of content in the iterable. Clear as mud? OK. Here’s an example:

   >>> for index, letter in enumerate(‘mud’):

   print(“{} {}”.format(index, letter))

   … 0 ‘m’

   ‘u’

   ‘d

   An optional, second parameter sets the starting point for the index produced. To populate the actual data, we’re starting the Excel rows produced (output\_rows) at 2. The headers already occupy Row . Their values are based on the DataFrame rows (df\_rows) which start at 0. If we adjust enumerate()’s starting point appropriately, we can avoid confusing ad hoc index pointer adjustments in-line with the code. [↑](#footnote-ref-6)