

Python Excels

Doing cool stuff with Python

<http://pythonexcels.com/>

<http://pythonexcels.com/basic-excel-driving-with-python/>

<http://pythonexcels.com/python-excel-mini-cookbook/>

<http://pythonexcels.com/mapping-excel-vb-macros-to-python/>

<http://pythonexcels.com/mapping-excel-vb-macros-to-python-revisited/>

Basic Excel Driving with Python

Posted on [September 29, 2009](#)

Now it's getting interesting. Reading and writing spreadsheets with XLRD and XLWT is sufficient for many tasks, and you don't even need a copy of Excel to do it. But to really open up your data and fully wring all the information possible from it, you'll need Excel and its powerful set of functions, pivot tables and charting.

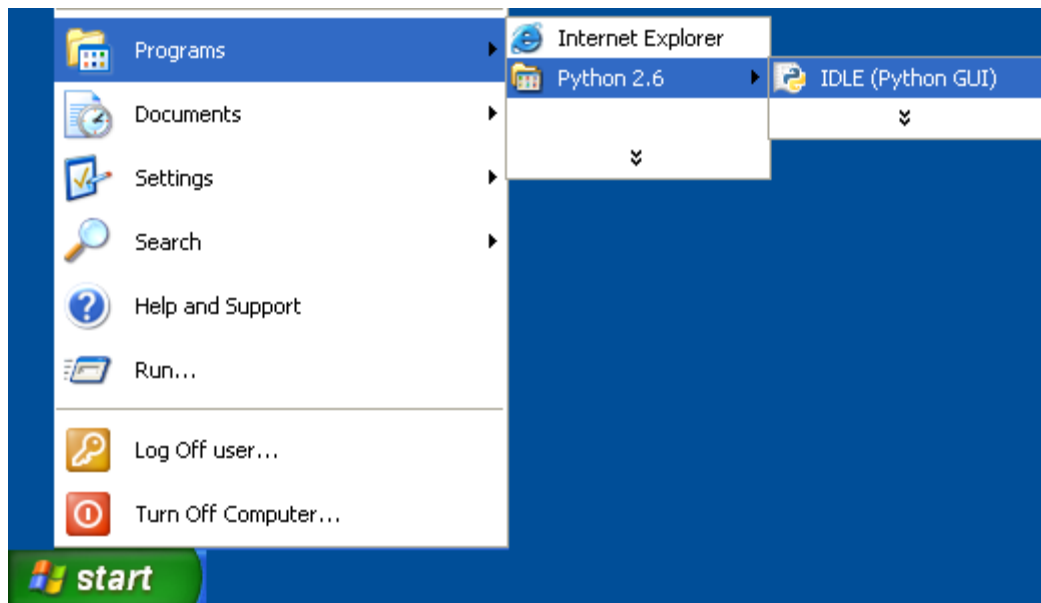
For starters, let's do some simple operations using Python to invoke Excel, add a spreadsheet, insert some data, then save the results to a spreadsheet file. You can play along at home by following my lead and entering the program text exactly as I've described below. My exercises and screen shots are done with Excel 2007, but all the commands presented here also work fine for Excel 2003. A prerequisite for this exercise is Python, the [Win32](#) module and a copy of Microsoft Excel.

Here is the [complete script](#) we'll be entering using IDLE, the Python interactive development tool.

```
#
# driving.py
#
import win32com.client as win32
excel = win32.gencache.EnsureDispatch('Excel.Application')
excel.Visible = True
wb = excel.Workbooks.Add()
ws = wb.Worksheets('Sheet1')
ws.Name = 'Built with Python'
ws.Cells(1,1).Value = 'Hello Excel'
print ws.Cells(1,1).Value
for i in range(1,5):
    ws.Cells(2,i).Value = i # Don't do this
ws.Range(ws.Cells(3,1),ws.Cells(3,4)).Value = [5,6,7,8]
ws.Range("A4:D4").Value = [i for i in range(9,13)]
ws.Cells(5,4).Formula = '=SUM(A2:D4)'
ws.Cells(5,4).Font.Size = 16
ws.Cells(5,4).Font.Bold = True
```

What follows is a step-by-step guide to entering this script and monitoring the result.

1. Open the Python IDLE interface from the Start menu



IDLE is the Python IDE built with the tkinter GUI toolkit, as quoted from the [Python IDLE documentation](#), and gives you an interactive interface to enter, run and save Python programs. IDLE isn't strictly necessary for this exercise, you could use any shell command window, or a tool such as [IPython](#) or the MS Windows command line interface.

2. Import the win32 module

```
>>> import win32com.client as win32
>>> |
```

If the import command was successful, you'll see the ">>>" prompt returned. If there was a problem, such as not having the win32 module installed correctly, you'll see `Import Error: Nomodule named win32com.client`. In that case, install the appropriate win32 module from the [web site](#).

3. Start Excel

The command `win32.gencache.EnsureDispatch('Excel.Application')` attaches to an Excel process that is already running, or starts Excel if it's not. If you see the ">>>" prompt, Excel has been started or linked successfully. At this point you won't see Excel, but if you check your task manager you can confirm that the process is running.

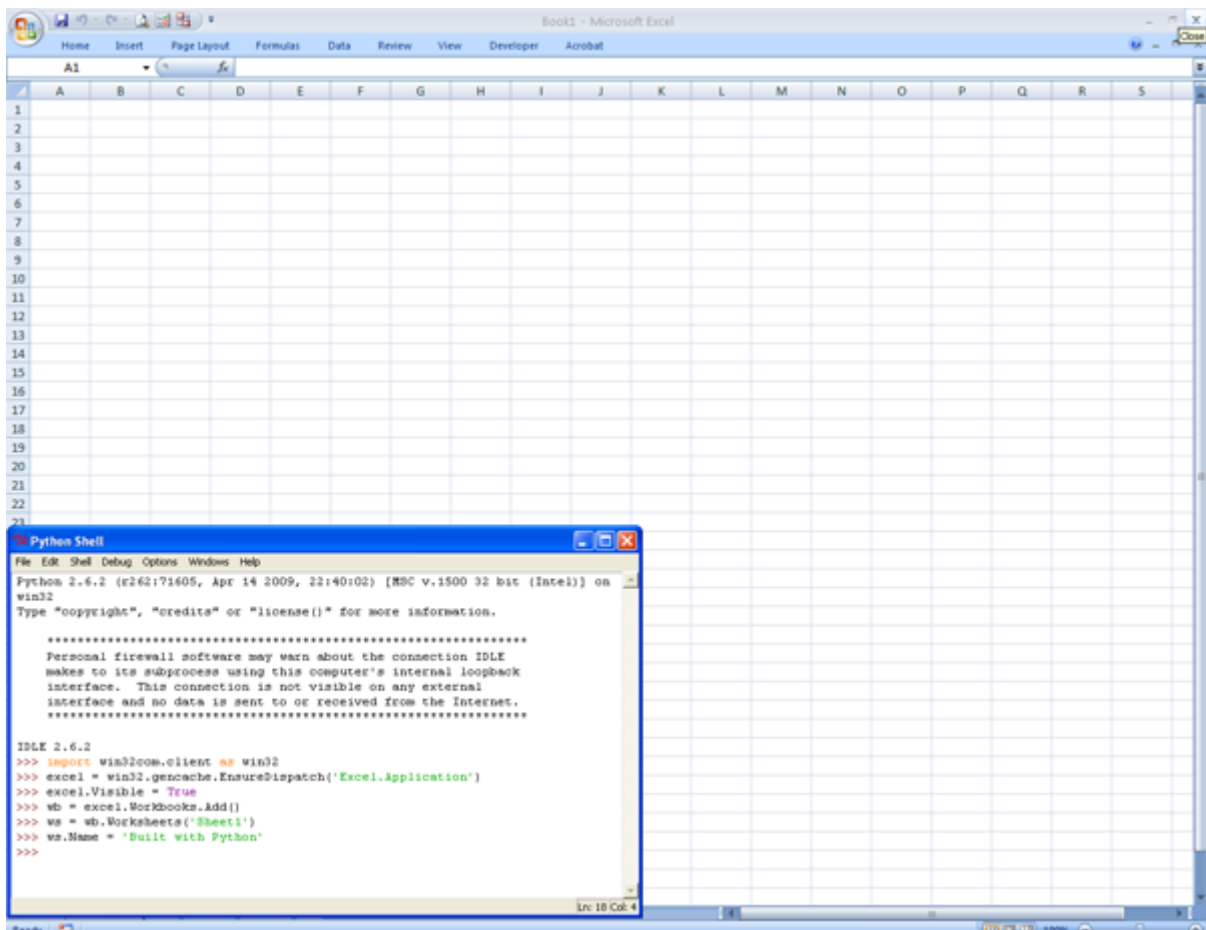
4. Make Excel Visible

Setting the Visible flag with `excel.Visible = True` makes the Excel window appear. At this point, Excel does not contain any workbooks or worksheets, we'll add those in the next step.

5. Add a workbook, select the sheet "Sheet1" and rename it

```
>>> wb = excel.Workbooks.Add()
>>> ws = wb.Worksheets('Sheet1')
>>> ws.Name = 'Built with Python'
```

Excel needs a workbook to serve as a container for the worksheets. A new workbook containing 3 sheets is added with command `wb = excel.Workbooks.Add()`. The command `ws = wb.Worksheets('Sheet1')` assigns `ws` to the sheet named `Sheet1`, and the command `ws.Name = 'Built with Python'` changes the name of `Sheet1` to “Built with Python”. Your screen should now look something like this:

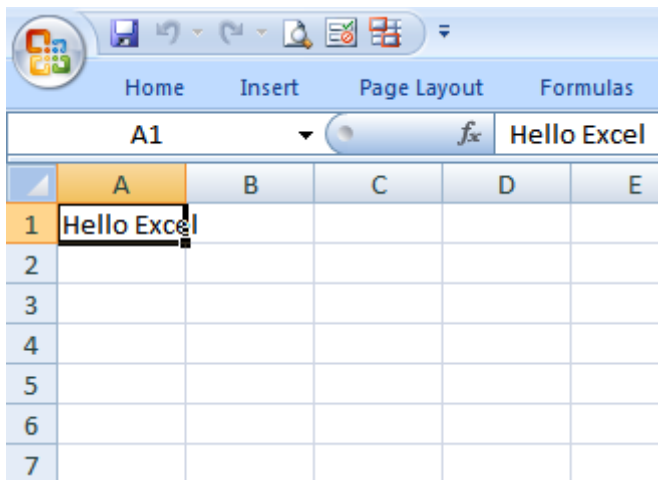


6. Add some text into the first cell

```
>>> ws.Cells(1,1).Value = 'Hello Excel'
>>> print ws.Cells(1,1).Value
Hello Excel
```

Now the setup is complete and you can add data to the spreadsheet. There are several options for addressing cells and blocks of data in Excel, I'll cover a few of them here. You can address individual cells with the `Cells(row, column).Value` pattern, where row and column are integer values representing the row and column location for the cell. Note that row and column counts begin from one, not zero. Use `.Value` to add text, numbers and date information to the cell and use `.Formula` for entering an Excel formula into the cell location.

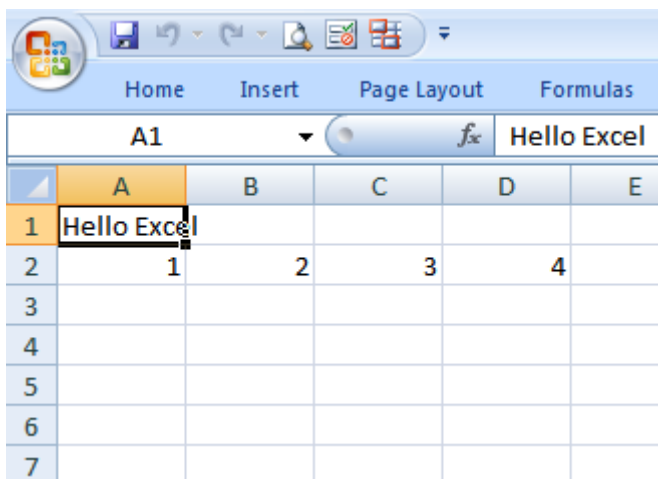
After typing these commands, you'll see the "Hello Excel" text in your Excel worksheet, and see the text printed in the IDLE window as well. Of course, Python can set values in the spreadsheet as well as query data from the spreadsheet.



7. Populate the second row with data by using a `for` loop

```
>>> for i in range(1,5):  
    ws.Cells(2,i).Value = i
```

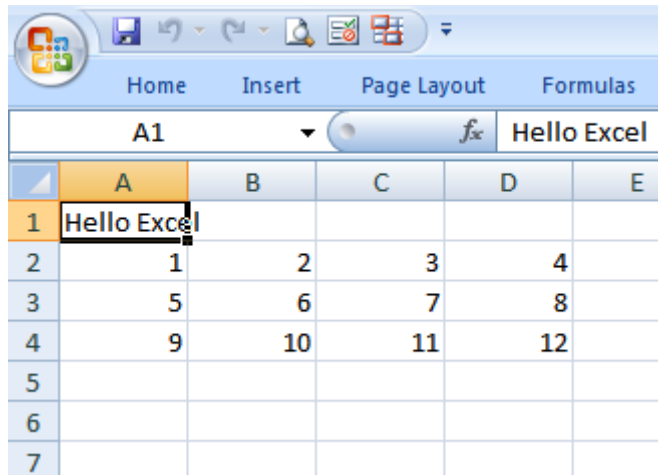
In many cases you'll have lists of data to insert into or extract from the worksheet. Wrapping the `Cells(row,column).Value` pattern with a loop seems like a natural approach, but in reality this maximizes the communication overhead between Python and Excel and results in very inefficient and slow code. It's much better to transfer lists than individual elements whenever possible as shown in the next section. After this command, your Excel spreadsheet will look like this:



8. Populate the third and fourth rows of data

```
>>> ws.Range(ws.Cells(3,1),ws.Cells(3,4)).Value = [5,6,7,8]  
>>> ws.Range("A4:D4").Value = [i for i in range(9,13)]
```

A better approach to populating or extracting blocks of data is to use the `Range().Value` pattern. With this construct you can efficiently transfer a one- or two-dimensional blocks of data. In the first example, cells (3,1) through (3,4) are assigned to the list `[5,6,7,8]`. The next line uses the Excel-style cell address “A4:D4” to assign the results of the operation `[i for i in range(9,13)]`. In some cases, it may be more intuitive to use the Excel-style naming. The Excel sheet now looks like this:

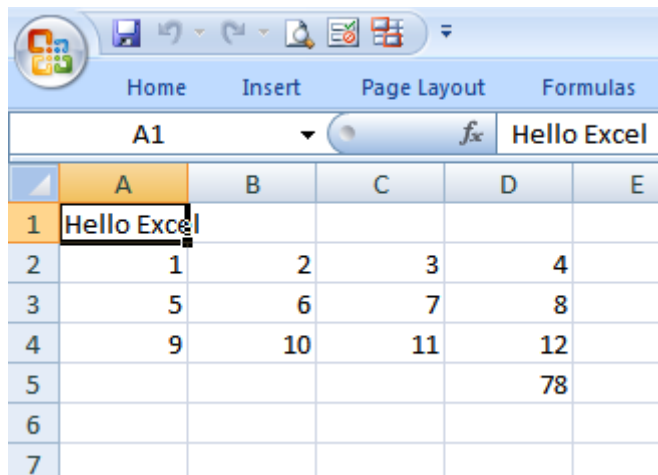


| | A | B | C | D | E |
|---|-------------|----|----|----|---|
| 1 | Hello Excel | | | | |
| 2 | 1 | 2 | 3 | 4 | |
| 3 | 5 | 6 | 7 | 8 | |
| 4 | 9 | 10 | 11 | 12 | |
| 5 | | | | | |
| 6 | | | | | |
| 7 | | | | | |

9. Assign a formula to sum the numbers just added

```
>>> ws.Cells(5,4).Formula = '=SUM(A2:D4)'
```

You can insert Excel formulas into cells using the `.Formula` pattern. The formula is the same as if you were to enter it in Excel: `=SUM(A2:D4)`. In this example, the sum of 12 numbers in rows 2,3 and 4 is generated. Your Excel sheet should now look like the screenshot below.

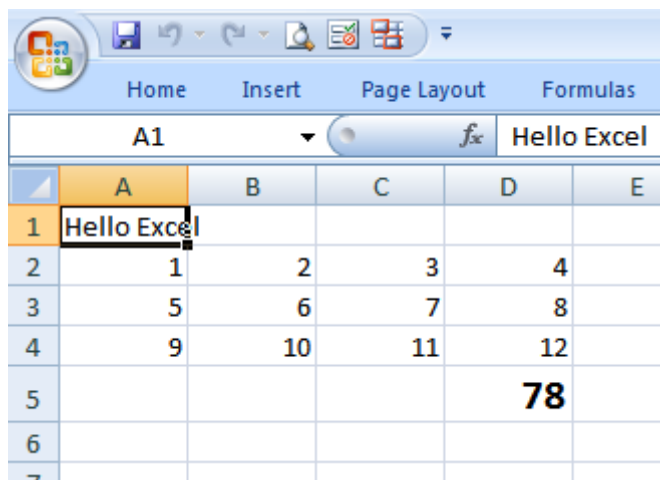


| | A | B | C | D | E |
|---|-------------|----|----|----|---|
| 1 | Hello Excel | | | | |
| 2 | 1 | 2 | 3 | 4 | |
| 3 | 5 | 6 | 7 | 8 | |
| 4 | 9 | 10 | 11 | 12 | |
| 5 | | | | 78 | |
| 6 | | | | | |
| 7 | | | | | |

10. Change the formatting of the formula cell **

```
>>> ws.Cells(5,4).Font.Size = 16
>>> ws.Cells(5,4).Font.Bold = True
```

As a final exercise, the format of the formula cell is changed to point size 16 with a bold typeface. You can change any of dozens of attributes for the various cells in the worksheet through Python. Your spreadsheet should now look like this.



| | A | B | C | D | E |
|---|-------------|----|----|-----------|---|
| 1 | Hello Excel | | | | |
| 2 | 1 | 2 | 3 | 4 | |
| 3 | 5 | 6 | 7 | 8 | |
| 4 | 9 | 10 | 11 | 12 | |
| 5 | | | | 78 | |
| 6 | | | | | |
| 7 | | | | | |

Hopefully you did this exercise interactively, typing the commands and monitoring the result in Excel. You can also cut to the chase and run this script to generate the result. When the script exits, you'll be left with an open Excel spreadsheet just as shown in the last screenshot above.

Prerequisites

- Python (refer to <http://www.python.org>)
- Win32 Python module (refer to <http://sourceforge.net/projects/pywin32>)
- Microsoft Excel

Source Files and Scripts

Source for the program and data text file are available at <http://github.com/pythonexcels/examples/tree/master>

References

[Core Python Programming](#)

Wesley Chun's book has a chapter on Programming Microsoft Office with Win32 COM

<http://groups.google.com/group/python-excel>

Though this group mainly covers questions on the excellent XLRD, XLWT and XLUTILS modules, there is also some discussion on interfacing to Excel using Win32 COM

[Stack Overflow](#)

Stack Overflow is a great resource for getting questions answered on a variety of programming topics, including Python

Thanks everyone — Dan

Python Excel Mini Cookbook

Posted on [October 5, 2009](#)

To get you started, I've illustrated a number of common tasks you can do with Python and Excel. Each program below is a self contained example, just copy it, paste it and run it. A few things to note:

- These examples were tested in Excel 2007, they should work fine in earlier versions as well after changing the extension of the file within the `wb.SaveAs()` statement from `.xlsx` to `.xls`
- If you're new to this, I recommend typing these examples by hand into IDLE, IPython or the Python interpreter, then watching the effect in Excel as you enter the commands. To make Excel visible add the line `excel.Visible = True` after the `excel = win32.gencache.EnsureDispatch('Excel.Application')` line in the script
- These are simple examples with no error checking. Make sure the output files doesn't exist before running the script. If the script crashes, it may leave a copy of Excel running in the background. Open the Windows Task Manager and kill the background Excel process to recover.
- These examples contain no optimization. You typically wouldn't use a `for` loop to iterate through data in individual cells, it's provided here for illustration only.

Open Excel, Add a Workbook

The [following script](#) simply invokes Excel, adds a workbook and saves the empty workbook.

```
#
# Add a workbook and save (Excel 2007)
# For older versions of excel, use the .xls file extension
#
import win32com.client as win32
excel = win32.gencache.EnsureDispatch('Excel.Application')
wb = excel.Workbooks.Add()
wb.SaveAs('add_a_workbook.xlsx')
excel.Application.Quit()
```

Open an Existing Workbook

[This script](#) opens an existing workbook and displays it (note the statement `excel.Visible = True`). The file `workbook1.xlsx` must already exist in your "My Documents" directory. You can also open spreadsheet files by specifying the full path to the file as shown below. Using `r` in the statement `r'C:\myfiles\excel\workbook2.xlsx'` automatically escapes the backslash characters and makes the file name a bit more concise.

```
#
# Open an existing workbook
#
import win32com.client as win32
excel = win32.gencache.EnsureDispatch('Excel.Application')
```

```
wb = excel.Workbooks.Open('workbook1.xlsx')
# Alternately, specify the full path to the workbook
# wb = excel.Workbooks.Open(r'C:\myfiles\excel\workbook2.xlsx')
excel.Visible = True
```

Add a Worksheet

[This script](#) creates a new workbook with three sheets, adds a fourth worksheet and names it MyNewSheet.

```
#
# Add a workbook, add a worksheet,
# name it 'MyNewSheet' and save
#
import win32com.client as win32
excel = win32.gencache.EnsureDispatch('Excel.Application')
wb = excel.Workbooks.Add()
ws = wb.Worksheets.Add()
ws.Name = "MyNewSheet"
wb.SaveAs('add_a_worksheet.xlsx')
excel.Application.Quit()
```

Ranges and Offsets

[This script](#) illustrates different techniques for addressing cells by using the `Cells()` and `Range()` operators. Individual cells can be addressed using `Cells(row, column)`, where `row` is the row number, `column` is the column number, both start from 1. Groups of cells can be addressed using `Range()`, where the argument in the parenthesis can be a single cell denoted by its textual name (eg "A2"), a group noted by a textual name with a colon (eg "A3:B4") or a group denoted with two `Cells()` identifiers (eg `ws.Cells(1,1),ws.Cells(2,2)`). The `Offset` method provides a way to address a cell based on a reference to another cell.

```
#
# Using ranges and offsets
#
import win32com.client as win32
excel = win32.gencache.EnsureDispatch('Excel.Application')
wb = excel.Workbooks.Add()
ws = wb.Worksheets("Sheet1")
ws.Cells(1,1).Value = "Cell A1"
ws.Cells(1,1).Offset(2,4).Value = "Cell D2"
ws.Range("A2").Value = "Cell A2"
ws.Range("A3:B4").Value = "A3:B4"
ws.Range("A6:B7,A9:B10").Value = "A6:B7,A9:B10"
wb.SaveAs('ranges_and_offsets.xlsx')
excel.Application.Quit()
```

Autofill Cell Contents

[This script](#) uses Excel's autofill capability to examine data in cells A1 and A2, then autofill the remaining column of cells through A10.

```
#
# Autofill cell contents
#
```

```

import win32com.client as win32
excel = win32.gencache.EnsureDispatch('Excel.Application')
wb = excel.Workbooks.Add()
ws = wb.Worksheets("Sheet1")
ws.Range("A1").Value = 1
ws.Range("A2").Value = 2
ws.Range("A1:A2").AutoFill(ws.Range("A1:A10"),win32.constants.xlFillDefault)
wb.SaveAs('autofill_cells.xlsx')
excel.Application.Quit()

```

Cell Color

[This script](#) illustrates adding an interior color to the cell using `Interior.ColorIndex`. Column A, rows 1 through 20 are filled with a number and assigned that `ColorIndex`.

```

#
# Add an interior color to cells
#
import win32com.client as win32
excel = win32.gencache.EnsureDispatch('Excel.Application')
wb = excel.Workbooks.Add()
ws = wb.Worksheets("Sheet1")
for i in range(1,21):
    ws.Cells(i,1).Value = i
    ws.Cells(i,1).Interior.ColorIndex = i
wb.SaveAs('cell_color.xlsx')
excel.Application.Quit()

```

Column Formatting

[This script](#) creates two columns of data, one narrow and one wide, then formats the column width with the `ColumnWidth` property. You can also use the `Columns.AutoFit()` function to autofit all columns in the spreadsheet.

```

#
# Set column widths
#
import win32com.client as win32
excel = win32.gencache.EnsureDispatch('Excel.Application')
wb = excel.Workbooks.Add()
ws = wb.Worksheets("Sheet1")
ws.Range("A1:A10").Value = "A"
ws.Range("B1:B10").Value = "This is a very long line of text"
ws.Columns(1).ColumnWidth = 1
ws.Range("B:B").ColumnWidth = 27
# Alternately, you can autofit all columns in the worksheet
# ws.Columns.AutoFit()
wb.SaveAs('column_widths.xlsx')
excel.Application.Quit()

```

Copying Data from Worksheet to Worksheet

[This script](#) uses the `FillAcrossSheets()` method to copy data from one location to all other worksheets in the workbook. Specifically, the data in the range A1:J10 is copied from Sheet1 to sheets Sheet2 and Sheet3.

```
#
# Copy data and formatting from a range of one worksheet
# to all other worksheets in a workbook
#
import win32com.client as win32
excel = win32.gencache.EnsureDispatch('Excel.Application')
wb = excel.Workbooks.Add()
ws = wb.Worksheets("Sheet1")
ws.Range("A1:J10").Formula = "=row()*column()"
wb.Worksheets.FillAcrossSheets(wb.Worksheets("Sheet1").Range("A1:J10"))
wb.SaveAs('copy_worksheet_to_worksheet.xlsx')
excel.Application.Quit()
```

Format Worksheet Cells

[This script](#) creates two columns of data, then formats the font type and font size used in the worksheet. Five different fonts and sizes are used, the numbers are formatted using a monetary format.

```
#
# Format cell font name and size, format numbers in monetary format
#
import win32com.client as win32
excel = win32.gencache.EnsureDispatch('Excel.Application')
wb = excel.Workbooks.Add()
ws = wb.Worksheets("Sheet1")

for i,font in enumerate(["Arial","Courier
New","Garamond","Georgia","Verdana"]):
    ws.Range(ws.Cells(i+1,1),ws.Cells(i+1,2)).Value = [font,i+i]
    ws.Range(ws.Cells(i+1,1),ws.Cells(i+1,2)).Font.Name = font
    ws.Range(ws.Cells(i+1,1),ws.Cells(i+1,2)).Font.Size = 12+i

ws.Range("A1:A5").HorizontalAlignment = win32.constants.xlRight
ws.Range("B1:B5").NumberFormat = "$###,##0.00"
ws.Columns.AutoFit()
wb.SaveAs('format_cells.xlsx')
excel.Application.Quit()
```

Setting Row Height

[This script](#) illustrates row height. Similar to column height, row height can be set with the `RowHeight` method. You can also use `AutoFit()` to automatically adjust the row height based on cell contents.

```
#
# Set row heights and align text within the cell
#
import win32com.client as win32
excel = win32.gencache.EnsureDispatch('Excel.Application')
wb = excel.Workbooks.Add()
ws = wb.Worksheets("Sheet1")
ws.Range("A1:A2").Value = "1 line"
ws.Range("B1:B2").Value = "Two\nlines"
ws.Range("C1:C2").Value = "Three\nlines\nhere"
ws.Range("D1:D2").Value = "This\nis\nfour\nlines"
ws.Rows(1).RowHeight = 60
ws.Range("2:2").RowHeight = 120
```

```
ws.Rows(1).VerticalAlignment = win32.constants.xlCenter
ws.Range("2:2").VerticalAlignment = win32.constants.xlCenter

# Alternately, you can autofit all rows in the worksheet
# ws.Rows.AutoFit()

wb.SaveAs('row_height.xlsx')
excel.Application.Quit()
```

Prerequisites

Python (refer to <http://www.python.org>)

Win32 Python module (refer to <http://sourceforge.net/projects/pywin32>)

Microsoft Excel (refer to <http://office.microsoft.com/excel>)

Source Files and Scripts

Source for the program and data text file are available
at <http://github.com/pythonexcels/examples>

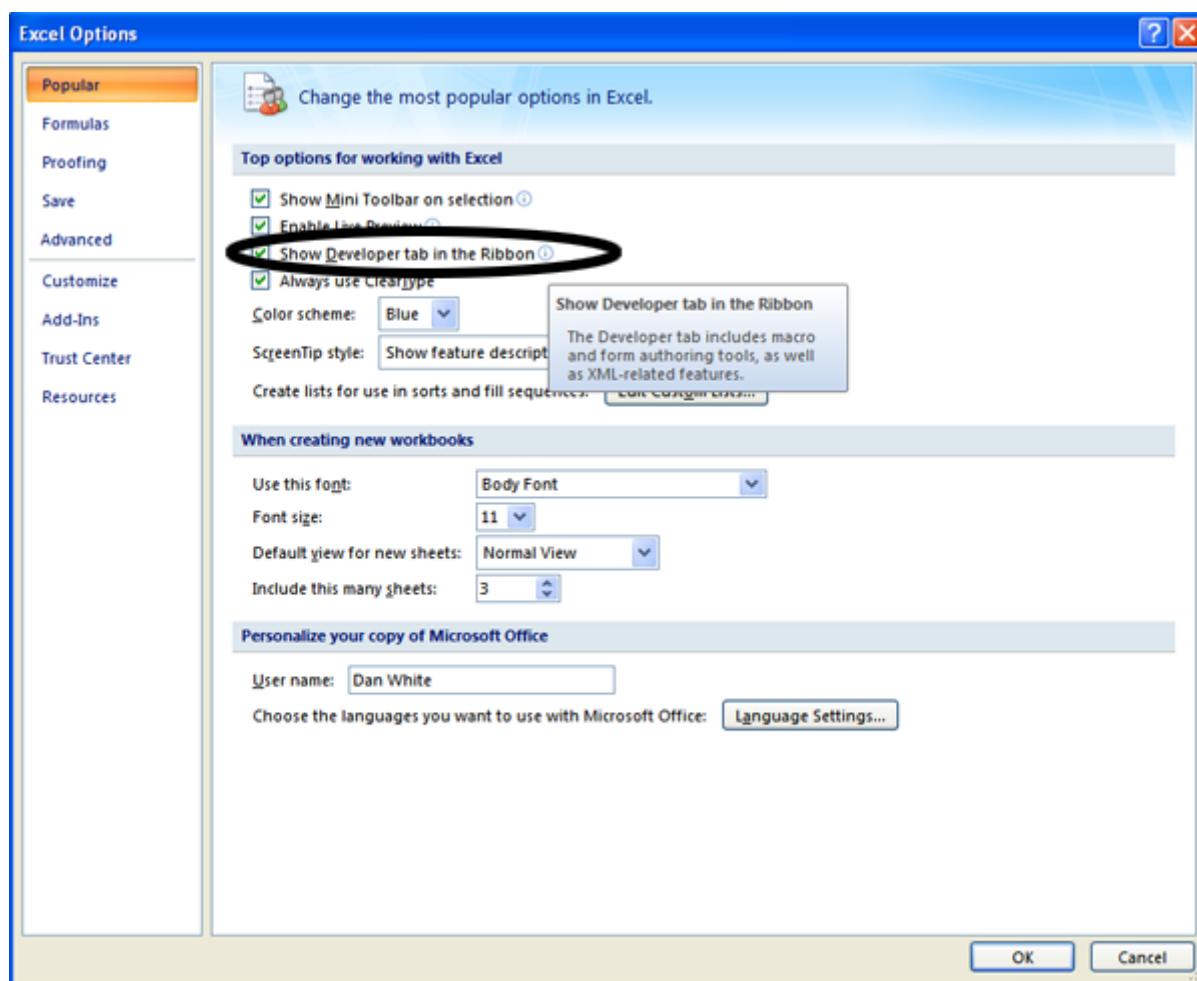
That's all for now, thanks — Dan

Mapping Excel VB Macros to Python

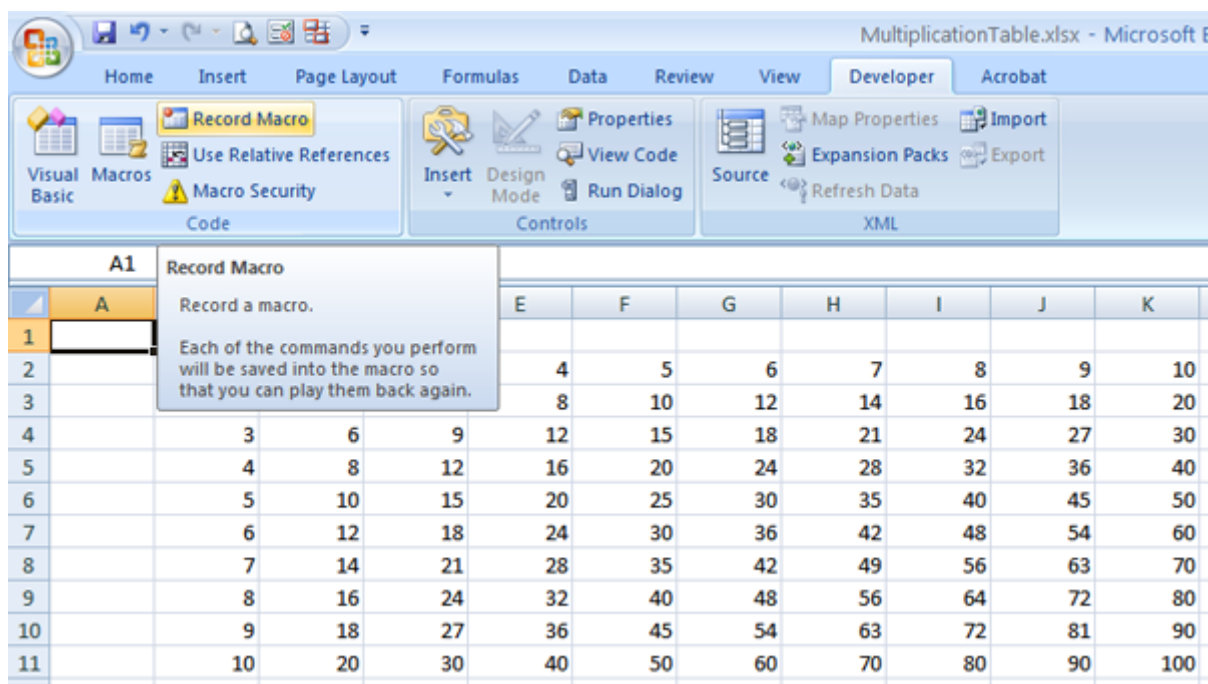
Posted on [October 12, 2009](#)

A handy feature in Excel is the ability to quickly record a Visual Basic (VB) macro and save it. It's also fairly simple to take a captured VB macro, tweak it slightly and use it in your Python scripts. I've used this capability dozens of times over the years to capture a sequence of operations that modify a spreadsheet and build a pivot table or chart, then integrate the macro into a Python script. It wasn't always apparent to me how I could take the macro and use it within Python or other scripting languages beside Visual Basic, but I now have a pretty good grasp and want to share the technique with you.

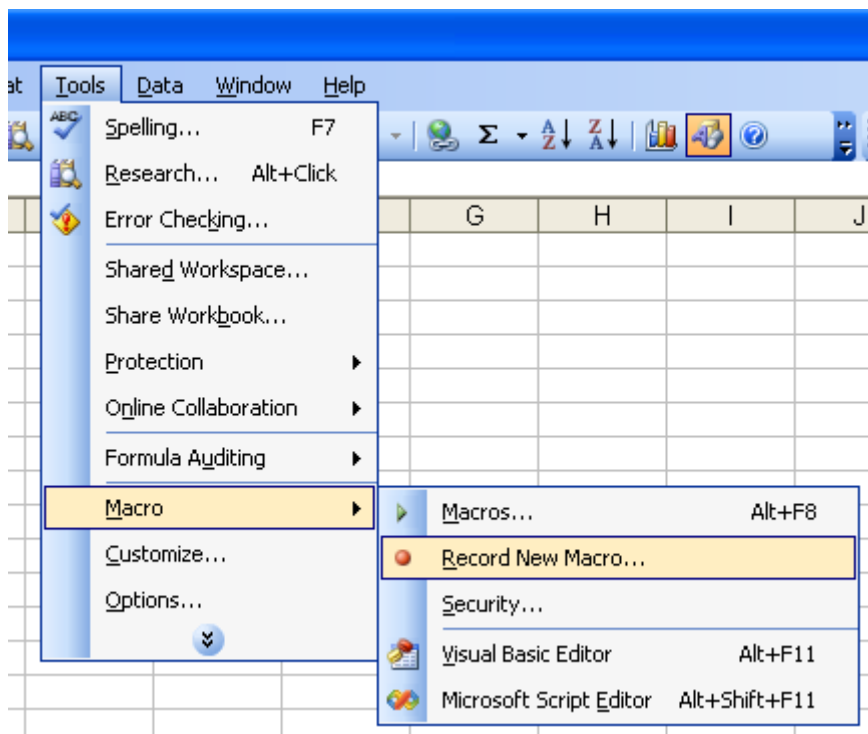
This post illustrates how to capture a simple set of operations in a macro, examine the Visual Basic macro, port it to Python and run it. I'm using the MultiplicationTable.xlsx file as a starting point, it's a simple 10×10 multiplication table that will be expanded and reformatted. The first step is to capture the macro in Excel using Record Macro. In Excel 2007 the Developer tab that contains the Record Macro button is turned off by default, you will need to enable it by selecting "Excel Options" from the ribbon menu, then select "Popular" in the left hand column and select the "Show Developer tab in the Ribbon" checkbox as shown here.



Starting with a simple spreadsheet containing a table of data, click on the “Developer” tab, then “Record Macro”.



If you're using an older version of Excel, select Tools->Macro->Record New Macro from the menu as shown here.



The goal is to expand the existing table to a 15×15 table, adjust the column width to make the table appear more square and save the new spreadsheet. Now that the macro is recording, the first step is to select the last row of data and expanding it by dragging it down an additional 5 rows. First, select the data:

MultiplicationTable.xlsx - Microsoft Excel

Home Insert Page Layout Formulas Data Review View Developer Acrobat

Visual Basic Macros Record Macro Use Relative References Macro Security Code

Insert Design Mode Properties View Code Run Dialog Controls

Source Map Properties Expansion Packs Refresh Data Import Export XML

| | A | B | C | D | E | F | G | H | I | J | K |
|----|---|----|----|----|----|----|----|----|----|----|-----|
| 1 | | | | | | | | | | | |
| 2 | | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| 3 | | 2 | 4 | 6 | 8 | 10 | 12 | 14 | 16 | 18 | 20 |
| 4 | | 3 | 6 | 9 | 12 | 15 | 18 | 21 | 24 | 27 | 30 |
| 5 | | 4 | 8 | 12 | 16 | 20 | 24 | 28 | 32 | 36 | 40 |
| 6 | | 5 | 10 | 15 | 20 | 25 | 30 | 35 | 40 | 45 | 50 |
| 7 | | 6 | 12 | 18 | 24 | 30 | 36 | 42 | 48 | 54 | 60 |
| 8 | | 7 | 14 | 21 | 28 | 35 | 42 | 49 | 56 | 63 | 70 |
| 9 | | 8 | 16 | 24 | 32 | 40 | 48 | 56 | 64 | 72 | 80 |
| 10 | | 9 | 18 | 27 | 36 | 45 | 54 | 63 | 72 | 81 | 90 |
| 11 | | 10 | 20 | 30 | 40 | 50 | 60 | 70 | 80 | 90 | 100 |
| 12 | | | | | | | | | | | |

then dragged to create 5 new rows of data.

MultiplicationTable.xlsx - Microsoft Excel

Home Insert Page Layout Formulas Data Review View Developer Acrobat

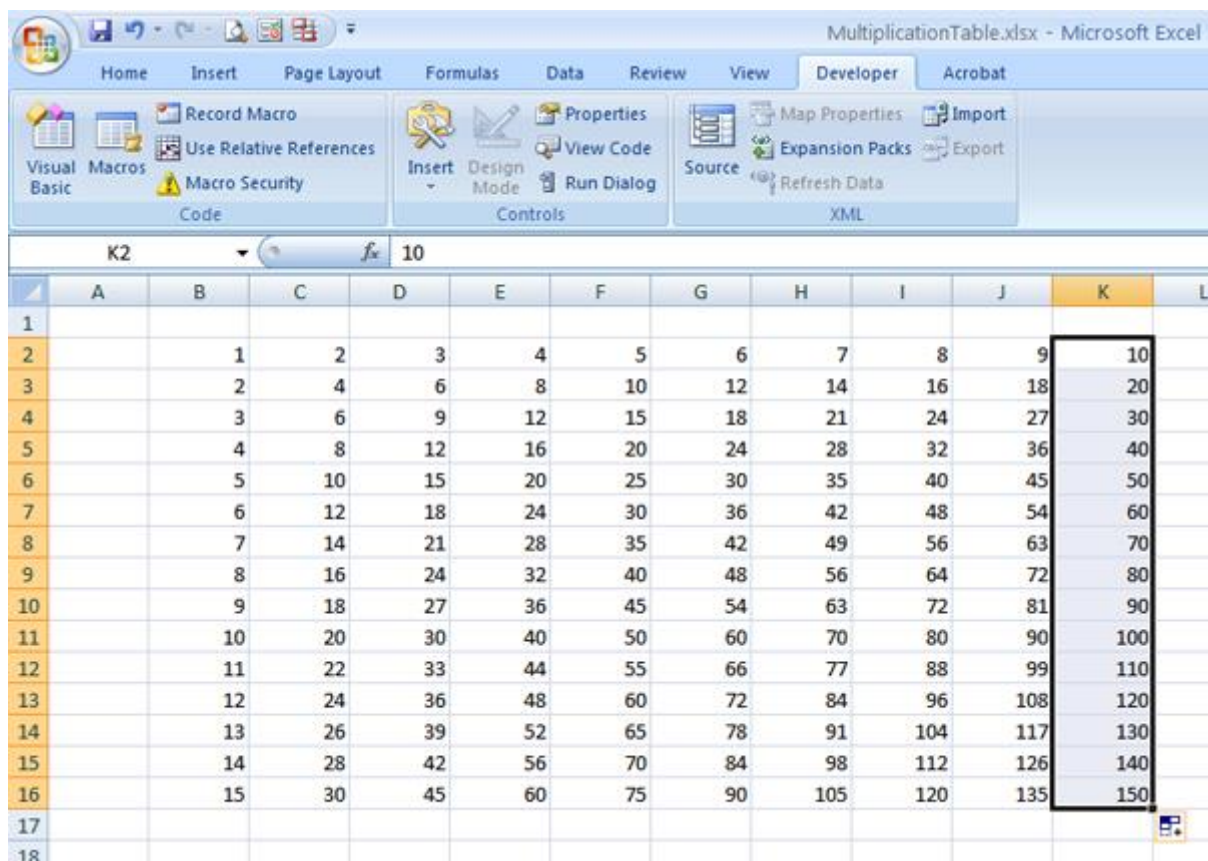
Visual Basic Macros Record Macro Use Relative References Macro Security Code

Insert Design Mode Properties View Code Run Dialog Controls

Source Map Properties Expansion Packs Refresh Data Import Export XML

| | A | B | C | D | E | F | G | H | I | J | K | L |
|----|---|----|----|----|----|----|----|----|----|----|-----|---|
| 1 | | | | | | | | | | | | |
| 2 | | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | |
| 3 | | 2 | 4 | 6 | 8 | 10 | 12 | 14 | 16 | 18 | 20 | |
| 4 | | 3 | 6 | 9 | 12 | 15 | 18 | 21 | 24 | 27 | 30 | |
| 5 | | 4 | 8 | 12 | 16 | 20 | 24 | 28 | 32 | 36 | 40 | |
| 6 | | 5 | 10 | 15 | 20 | 25 | 30 | 35 | 40 | 45 | 50 | |
| 7 | | 6 | 12 | 18 | 24 | 30 | 36 | 42 | 48 | 54 | 60 | |
| 8 | | 7 | 14 | 21 | 28 | 35 | 42 | 49 | 56 | 63 | 70 | |
| 9 | | 8 | 16 | 24 | 32 | 40 | 48 | 56 | 64 | 72 | 80 | |
| 10 | | 9 | 18 | 27 | 36 | 45 | 54 | 63 | 72 | 81 | 90 | |
| 11 | | 10 | 20 | 30 | 40 | 50 | 60 | 70 | 80 | 90 | 100 | |
| 12 | | | | | | | | | | | | |
| 13 | | | | | | | | | | | | |
| 14 | | | | | | | | | | | | |
| 15 | | | | | | | | | | | | |
| 16 | | | | | | | | | | | | |
| 17 | | | | | | | | | | | | |
| 18 | | | | | | | | | | | | |

Do the same select and drag operation for the last column of data to create 5 new columns.



MultiplicationTable.xlsx - Microsoft Excel

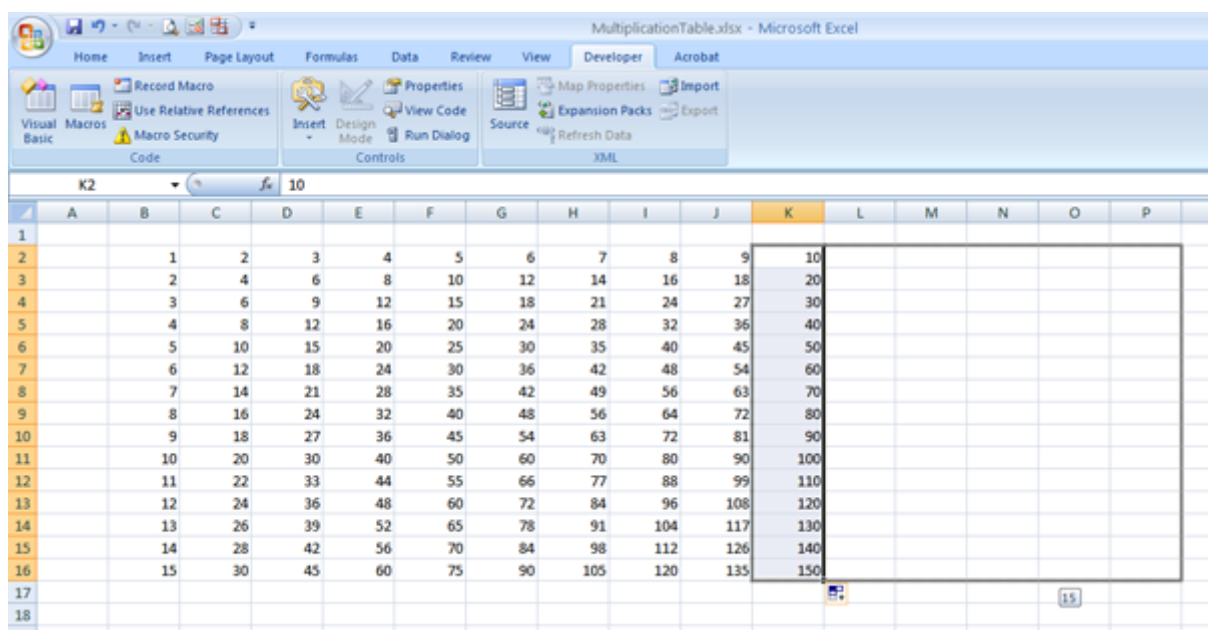
Home Insert Page Layout Formulas Data Review View Developer Acrobat

Visual Basic Macros Record Macro Use Relative References Macro Security Code

Insert Design Mode Properties View Code Run Dialog Controls

Source Map Properties Import Expansion Packs Export Refresh Data XML

| | A | B | C | D | E | F | G | H | I | J | K |
|----|---|----|----|----|----|----|----|-----|-----|-----|-----|
| 1 | | | | | | | | | | | |
| 2 | | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| 3 | | 2 | 4 | 6 | 8 | 10 | 12 | 14 | 16 | 18 | 20 |
| 4 | | 3 | 6 | 9 | 12 | 15 | 18 | 21 | 24 | 27 | 30 |
| 5 | | 4 | 8 | 12 | 16 | 20 | 24 | 28 | 32 | 36 | 40 |
| 6 | | 5 | 10 | 15 | 20 | 25 | 30 | 35 | 40 | 45 | 50 |
| 7 | | 6 | 12 | 18 | 24 | 30 | 36 | 42 | 48 | 54 | 60 |
| 8 | | 7 | 14 | 21 | 28 | 35 | 42 | 49 | 56 | 63 | 70 |
| 9 | | 8 | 16 | 24 | 32 | 40 | 48 | 56 | 64 | 72 | 80 |
| 10 | | 9 | 18 | 27 | 36 | 45 | 54 | 63 | 72 | 81 | 90 |
| 11 | | 10 | 20 | 30 | 40 | 50 | 60 | 70 | 80 | 90 | 100 |
| 12 | | 11 | 22 | 33 | 44 | 55 | 66 | 77 | 88 | 99 | 110 |
| 13 | | 12 | 24 | 36 | 48 | 60 | 72 | 84 | 96 | 108 | 120 |
| 14 | | 13 | 26 | 39 | 52 | 65 | 78 | 91 | 104 | 117 | 130 |
| 15 | | 14 | 28 | 42 | 56 | 70 | 84 | 98 | 112 | 126 | 140 |
| 16 | | 15 | 30 | 45 | 60 | 75 | 90 | 105 | 120 | 135 | 150 |
| 17 | | | | | | | | | | | |
| 18 | | | | | | | | | | | |



MultiplicationTable.xlsx - Microsoft Excel

Home Insert Page Layout Formulas Data Review View Developer Acrobat

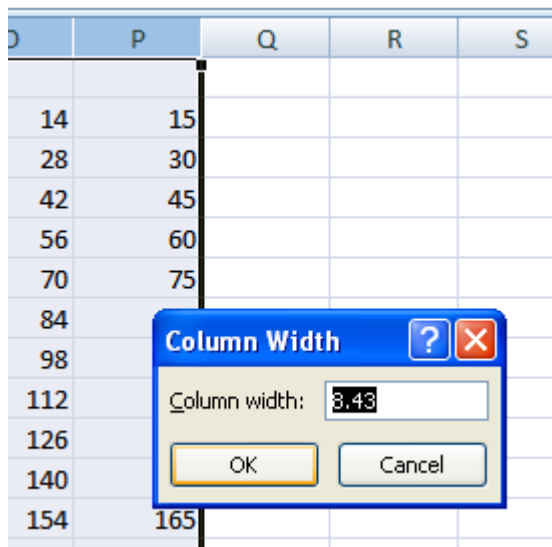
Visual Basic Macros Record Macro Use Relative References Macro Security Code

Insert Design Mode Properties View Code Run Dialog Controls

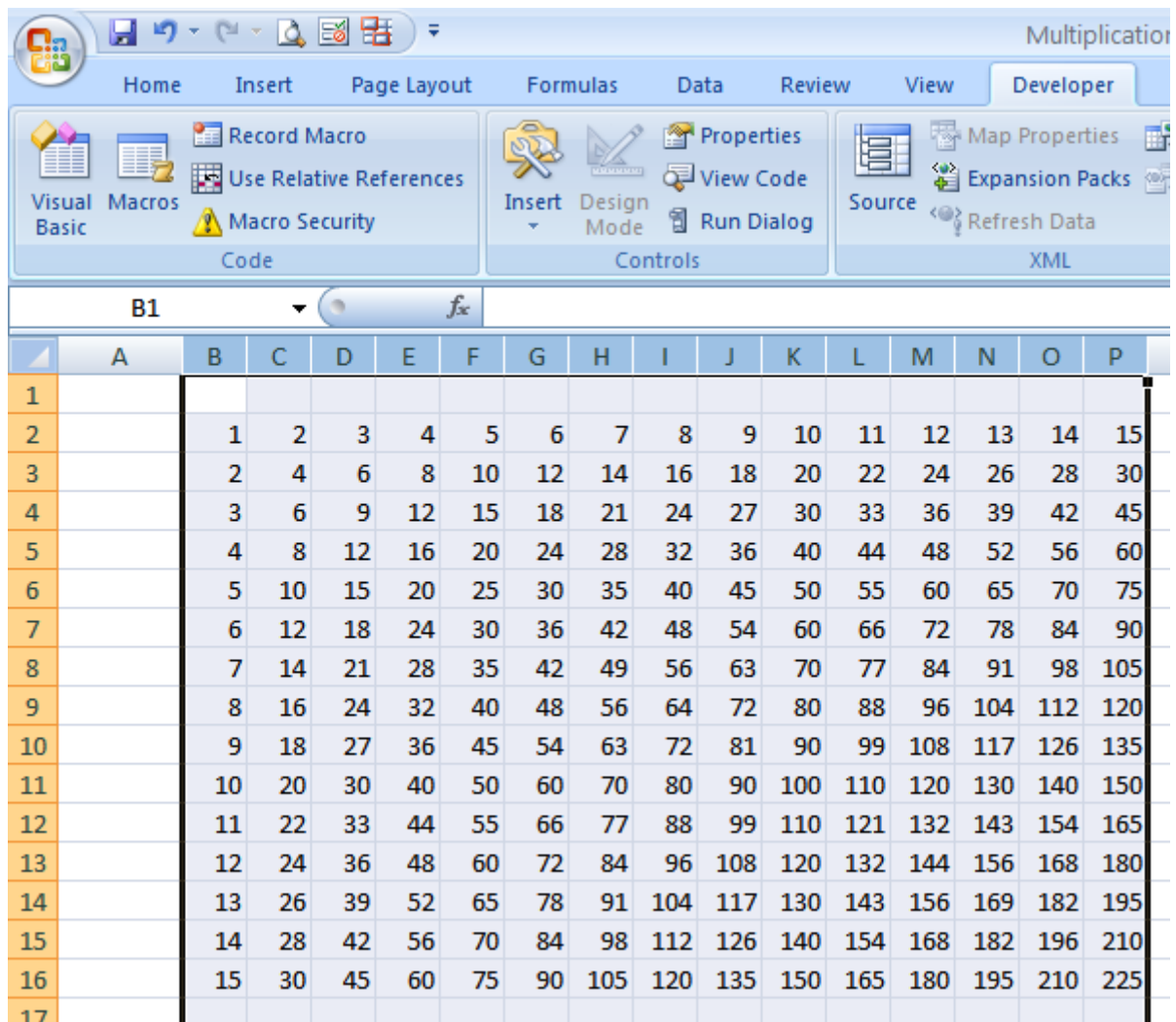
Source Map Properties Import Expansion Packs Export Refresh Data XML

| | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P |
|----|---|----|----|----|----|----|----|-----|-----|-----|-----|---|---|---|---|---|
| 1 | | | | | | | | | | | | | | | | |
| 2 | | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | | | | | |
| 3 | | 2 | 4 | 6 | 8 | 10 | 12 | 14 | 16 | 18 | 20 | | | | | |
| 4 | | 3 | 6 | 9 | 12 | 15 | 18 | 21 | 24 | 27 | 30 | | | | | |
| 5 | | 4 | 8 | 12 | 16 | 20 | 24 | 28 | 32 | 36 | 40 | | | | | |
| 6 | | 5 | 10 | 15 | 20 | 25 | 30 | 35 | 40 | 45 | 50 | | | | | |
| 7 | | 6 | 12 | 18 | 24 | 30 | 36 | 42 | 48 | 54 | 60 | | | | | |
| 8 | | 7 | 14 | 21 | 28 | 35 | 42 | 49 | 56 | 63 | 70 | | | | | |
| 9 | | 8 | 16 | 24 | 32 | 40 | 48 | 56 | 64 | 72 | 80 | | | | | |
| 10 | | 9 | 18 | 27 | 36 | 45 | 54 | 63 | 72 | 81 | 90 | | | | | |
| 11 | | 10 | 20 | 30 | 40 | 50 | 60 | 70 | 80 | 90 | 100 | | | | | |
| 12 | | 11 | 22 | 33 | 44 | 55 | 66 | 77 | 88 | 99 | 110 | | | | | |
| 13 | | 12 | 24 | 36 | 48 | 60 | 72 | 84 | 96 | 108 | 120 | | | | | |
| 14 | | 13 | 26 | 39 | 52 | 65 | 78 | 91 | 104 | 117 | 130 | | | | | |
| 15 | | 14 | 28 | 42 | 56 | 70 | 84 | 98 | 112 | 126 | 140 | | | | | |
| 16 | | 15 | 30 | 45 | 60 | 75 | 90 | 105 | 120 | 135 | 150 | | | | | |
| 17 | | | | | | | | | | | | | | | | |
| 18 | | | | | | | | | | | | | | | | |

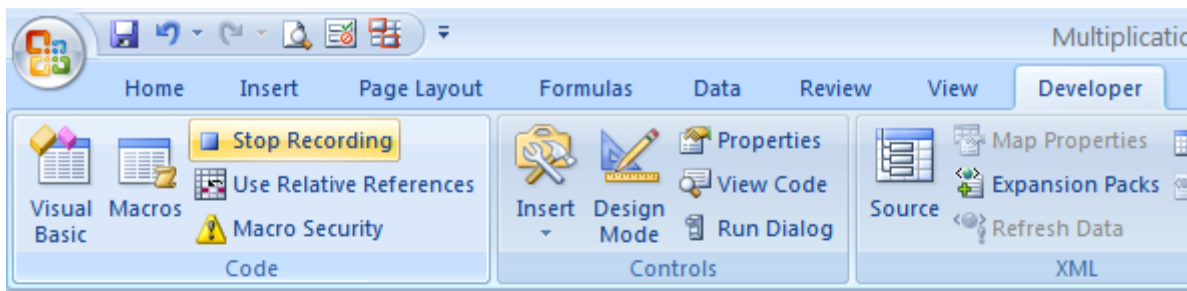
Now you have a 15×15 multiplication table. To resize the columns, select the headers for columns B through P, click the right mouse and select “Column Width”.



Enter “4” as the new column width and click OK. The spreadsheet will now look like this:



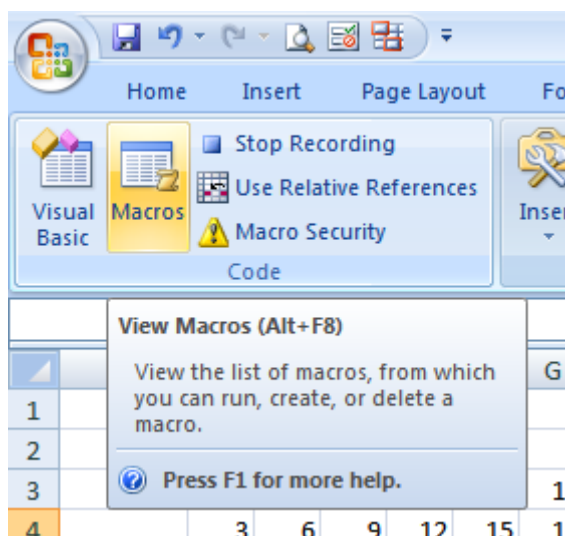
Now stop capturing the macro by clicking on Stop Recording



| | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P |
|----|---|----|----|----|----|----|----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| 1 | | | | | | | | | | | | | | | | |
| 2 | | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
| 3 | | 2 | 4 | 6 | 8 | 10 | 12 | 14 | 16 | 18 | 20 | 22 | 24 | 26 | 28 | 30 |
| 4 | | 3 | 6 | 9 | 12 | 15 | 18 | 21 | 24 | 27 | 30 | 33 | 36 | 39 | 42 | 45 |
| 5 | | 4 | 8 | 12 | 16 | 20 | 24 | 28 | 32 | 36 | 40 | 44 | 48 | 52 | 56 | 60 |
| 6 | | 5 | 10 | 15 | 20 | 25 | 30 | 35 | 40 | 45 | 50 | 55 | 60 | 65 | 70 | 75 |
| 7 | | 6 | 12 | 18 | 24 | 30 | 36 | 42 | 48 | 54 | 60 | 66 | 72 | 78 | 84 | 90 |
| 8 | | 7 | 14 | 21 | 28 | 35 | 42 | 49 | 56 | 63 | 70 | 77 | 84 | 91 | 98 | 105 |
| 9 | | 8 | 16 | 24 | 32 | 40 | 48 | 56 | 64 | 72 | 80 | 88 | 96 | 104 | 112 | 120 |
| 10 | | 9 | 18 | 27 | 36 | 45 | 54 | 63 | 72 | 81 | 90 | 99 | 108 | 117 | 126 | 135 |
| 11 | | 10 | 20 | 30 | 40 | 50 | 60 | 70 | 80 | 90 | 100 | 110 | 120 | 130 | 140 | 150 |
| 12 | | 11 | 22 | 33 | 44 | 55 | 66 | 77 | 88 | 99 | 110 | 121 | 132 | 143 | 154 | 165 |
| 13 | | 12 | 24 | 36 | 48 | 60 | 72 | 84 | 96 | 108 | 120 | 132 | 144 | 156 | 168 | 180 |
| 14 | | 13 | 26 | 39 | 52 | 65 | 78 | 91 | 104 | 117 | 130 | 143 | 156 | 169 | 182 | 195 |
| 15 | | 14 | 28 | 42 | 56 | 70 | 84 | 98 | 112 | 126 | 140 | 154 | 168 | 182 | 196 | 210 |
| 16 | | 15 | 30 | 45 | 60 | 75 | 90 | 105 | 120 | 135 | 150 | 165 | 180 | 195 | 210 | 225 |

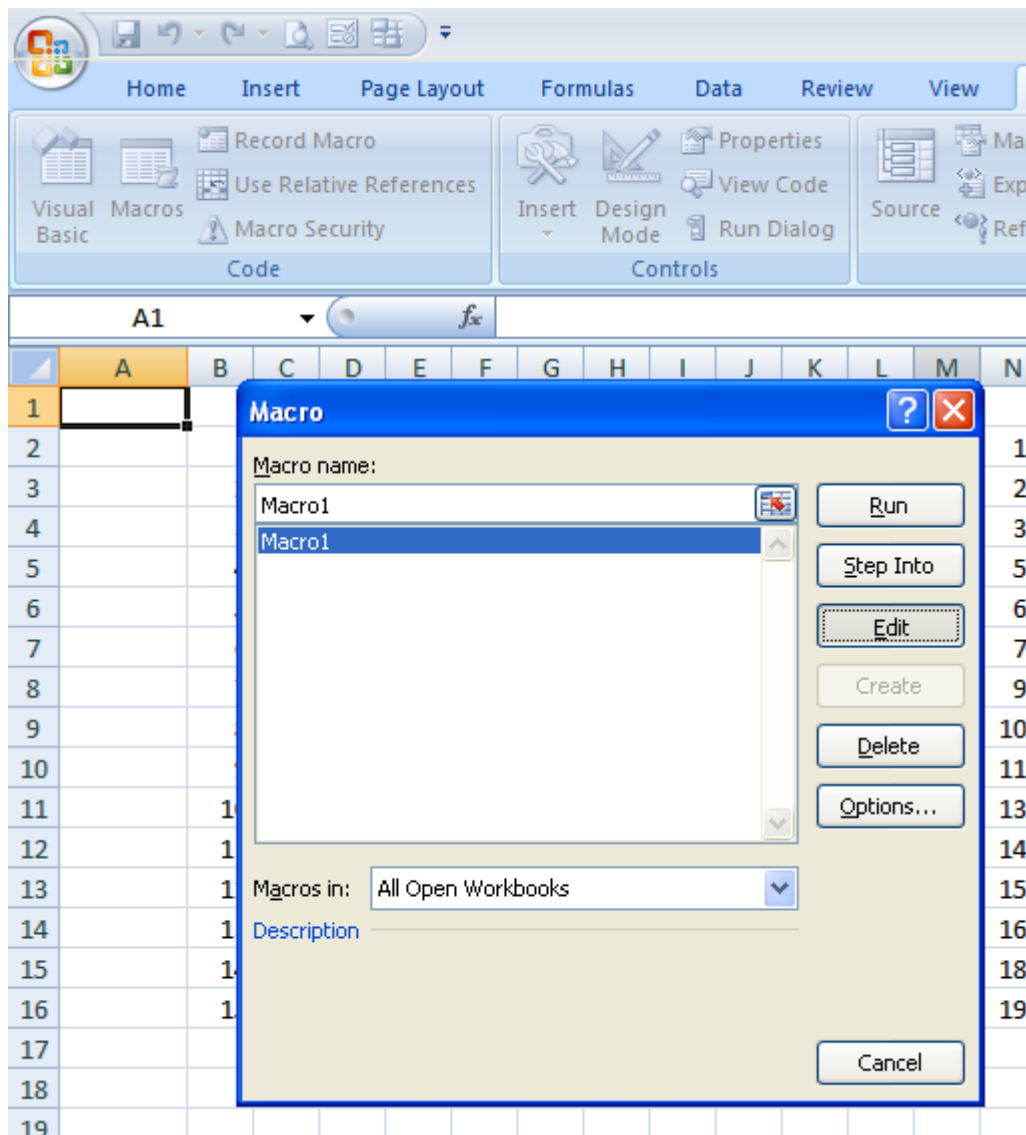
If you're using an older version of Excel, select Tools->Macro->Stop Recording from the menu bar.

To view the macro, click on the View Macros button



For older versions of Excel, select Tools->Macro->Macros

Select the macro you just recorded (this should be Macro1, but if you were experimenting you may have other macros, so select the highest numbered macro) and click Edit.



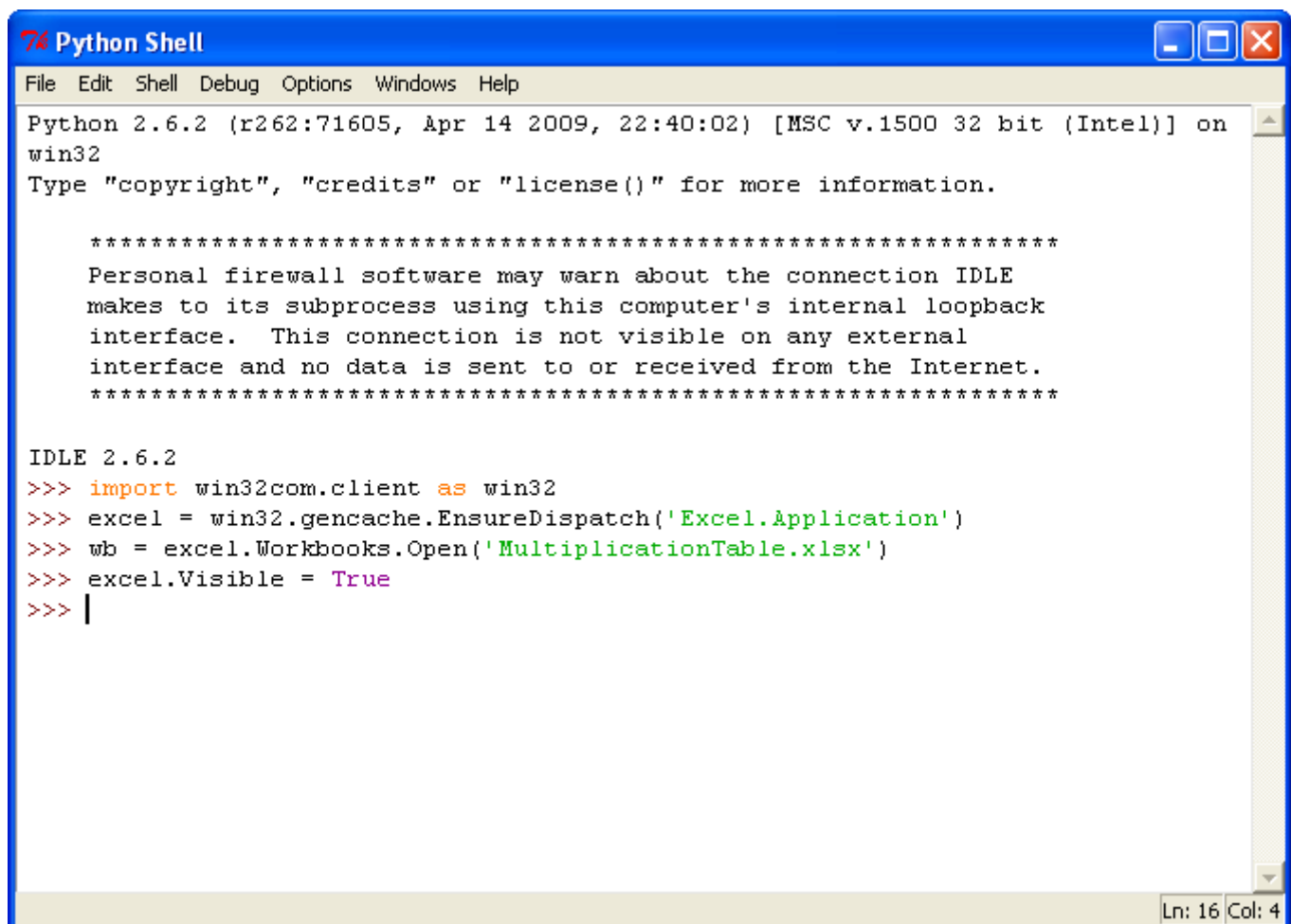
This will open your macro in the Microsoft Visual Basic GUI, and it should look something like this

```
Sub Macro1()  
'  
' Macro1 Macro  
'  
'  
  
    Range("B11:K11").Select  
    Selection.AutoFill Destination:=Range("B11:K16"), Type:=xlFillDefault  
    Range("B11:K16").Select  
    Range("K2:K16").Select  
    Selection.AutoFill Destination:=Range("K2:P16"), Type:=xlFillDefault  
    Range("K2:P16").Select  
    Columns("B:P").Select  
    Selection.ColumnWidth = 4  
End Sub
```

Don't worry if there are some extra or redundant lines in your macro, they can be removed as the script is ported. Now we're ready to fire up Python and integrate this macro into a script.

Porting

To get started, start the Python Integrated Development Environment (IDLE), then open the spreadsheet with the 10×10 multiplication table by entering the following four commands (make sure the file "MultiplicationTable.xlsx" is in your My Documents folder.

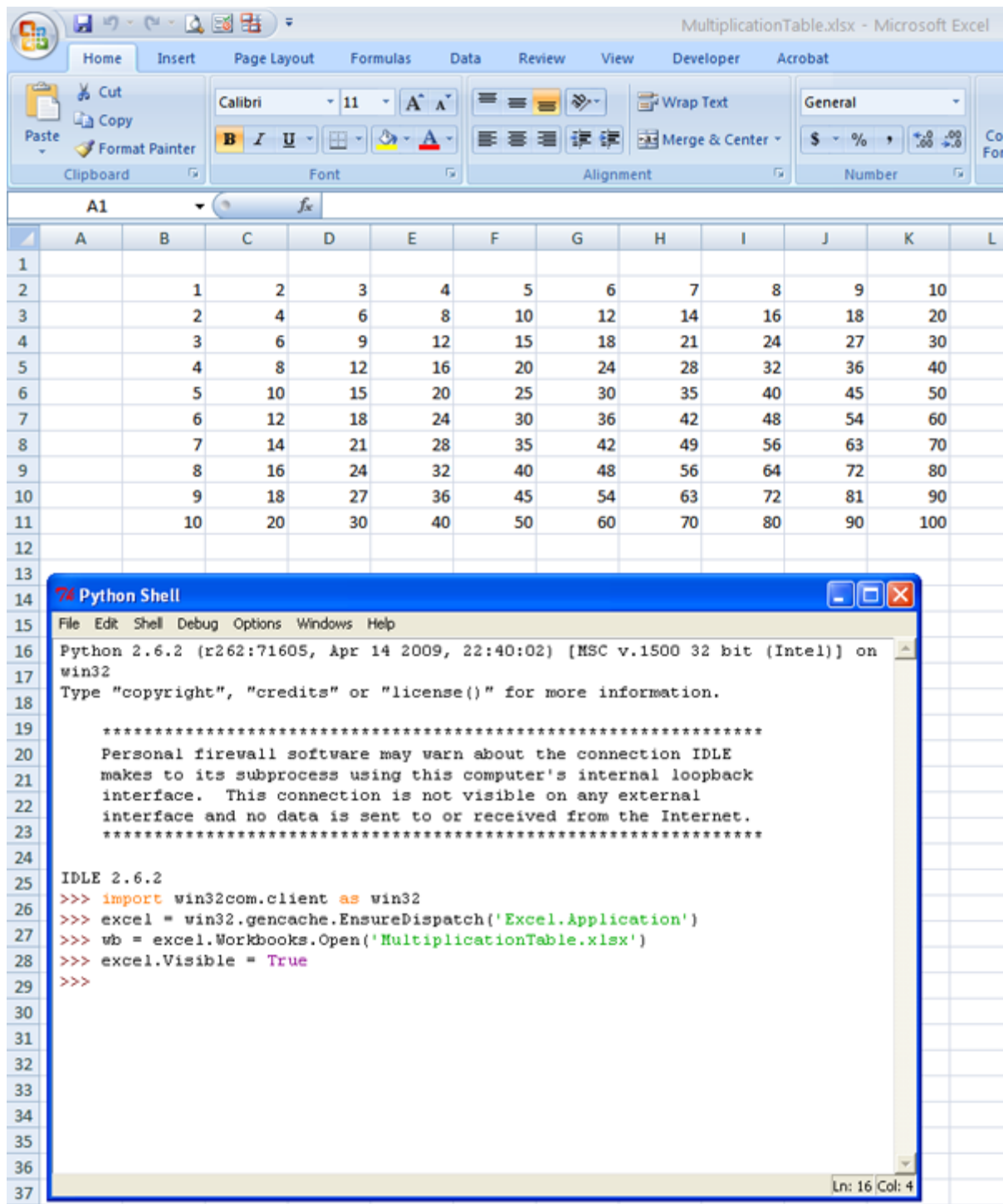


```
Python Shell
File Edit Shell Debug Options Windows Help
Python 2.6.2 (r262:71605, Apr 14 2009, 22:40:02) [MSC v.1500 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.

*****
Personal firewall software may warn about the connection IDLE
makes to its subprocess using this computer's internal loopback
interface. This connection is not visible on any external
interface and no data is sent to or received from the Internet.
*****

IDLE 2.6.2
>>> import win32com.client as win32
>>> excel = win32.gencache.EnsureDispatch('Excel.Application')
>>> wb = excel.Workbooks.Open('MultiplicationTable.xlsx')
>>> excel.Visible = True
>>> |
```

Your screen should now look like this:



These are boilerplate commands you'll be using in each exercise to invoke and interface to Excel. The first two commands, `import win32com.client as win32`, and `excel = win32.gencache.EnsureDispatch('Excel.Application')`, import the win32 module and open the Excel process. The command `wb = excel.Workbooks.Open('MultiplicationTable.xlsx')` opens the worksheet. In general, you'll need a `excel.Workbooks.Open()` or `excel.Workbooks.Add()` command to open an existing workbook or create a new workbook. The command `excel.Visible = True` makes Excel visible on the screen, rather than running as a hidden process in the background.

Looking at the Macro1 macro, the first command is `Range("B11:K11").Select`. The `Range` variable name is within the context of the `Worksheet`, so you need to create a container for operations on the worksheet. The command `ws = wb.Worksheets('Sheet1')` will do the trick.

```
>>> ws = wb.Worksheet('Sheet1')

Traceback (most recent call last):
  File "<pyshell#4>", line 1, in <module>
    ws = wb.Worksheet('Sheet1')
  File "C:\Python26\lib\site-packages\win32com\client\__init__.py", line 502, in
__getattr__
    if d is not None: return getattr(d, attr)
  File "C:\Python26\lib\site-packages\win32com\client\__init__.py", line 462, in
__getattr__
    raise AttributeError("%s' object has no attribute '%s'" % (repr(self), attr
))
AttributeError: '<win32com.gen_py.Microsoft Excel 12.0 Object Library._Workbook
instance at 0x16917248>' object has no attribute 'Worksheet'
>>> wb
<win32com.gen_py.None.Workbook>
>>> excel
<win32com.gen_py.Microsoft Excel 12.0 Object Library._Application instance at 0x
16663016>
>>> ws = wb.Worksheets('Sheet1')
>>> |
```

Ln: 31 Col: 4

If you noticed, I made a typo when entering the command and typed `Worksheet` instead of `Worksheets`. Don't panic if you make a mistake as I did, in most cases you can simply retype the correct command and continue on.

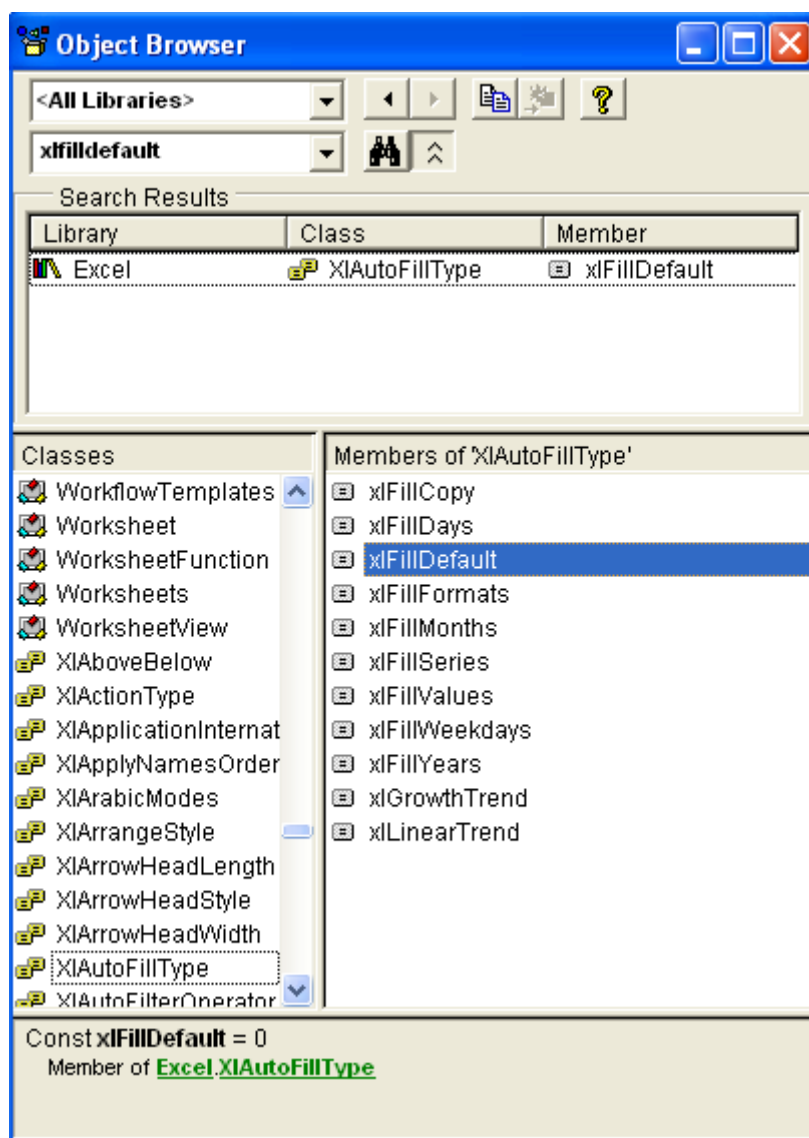
Once the variable pointing to the worksheet is defined, append the macro command to `ws`. and try it. Note that `Select` is a function and requires the open and close parenthesis pair in order to operate correctly. This pattern may be used for every `Range().Select` line in the macro.

```
>>> ws.Range("B11:K11").Select()
True
>>> |
```

Ln: 33 Col: 4

If you bring the worksheet to the foreground, you'll see that the range B11:K11 has been selected. The next task is to autofill the 5 rows below using the `Selection.AutoFillDestination:=Range("B11:K16"), Type:=xlFillDefault` construct. `Selection` is a method at the Excel Application level, you need to precede it with `excel.` in this example. The arguments `Destination:=Range("B11:K16"), Type:=xlFillDefault` must be provided to the function, either using the keyword arguments `Destination` and `Type`, or by using positional notation. To make your programs as robust as possible, you should include the keywords, but it's not strictly required and I don't use that pattern in this example.

The definition for the constant `xlFillDefault` is contained in `win32.constants`, you can access this value by specifying `win32.constants.xlFillDefault`. I've seen many examples where the developer replaces this with the actual value (0 in this case). My preference is to avoid replacing Excel variables with numbers in my scripts, I believe that including the variable names increases the clarity of the script. My preference is to use the fully specified name wherever possible, but if you have to replace the variable with the actual value, you can always use the Object Browser in the VB window to figure out the correct value (open the Object Browser by pressing F2, or by selecting View->Object Browser from the menu in the VB window).



Combining these translations, the full Python command is `excel.Selection.AutoFill(Destination=ws.Range("B11:K16"), Type=win32.constants.xlFillDefault)`, or `excel.Selection.AutoFill(ws.Range("B11:K16"), win32.constants.xlFillDefault)` as I've used in the example.


```
>>> excel.Selection.AutoFill(ws.Range('B11:K16'), win32.constants.xlFillDefault)
True
>>> |
```

Ln: 35 Col: 4

Occasionally you'll make a mistake when capturing a macro and record extraneous, unnecessary commands. The command `Range("B11:K16").Select` isn't needed and can be ignored. The next two macro commands, `Range("K2:K16").Select` and `Selection.AutoFill Destination:=Range("K2:P16"), Type:=xlFillDefault`, are translated in the same way as the `Select` and `AutoFill` commands discussed earlier.

The commands `Range("K2:K16").Select` and `Selection.AutoFill Destination:=Range("K2:P16"), Type:=xlFillDefault` are translated in the same fashion as the earlier `Select` and `AutoFill` commands as shown below.

```
>>> ws.Range("K2:K16").Select()
True
>>> excel.Selection.AutoFill(ws.Range("K2:P16"), win32.constants.xlFillDefault)
True
>>> |
```

Ln: 39 Col: 4

The worksheet is now expanded to the full 15×15 table and looks like this:

| | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P |
|----|---|----|----|----|----|----|----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| 1 | | | | | | | | | | | | | | | | |
| 2 | | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
| 3 | | 2 | 4 | 6 | 8 | 10 | 12 | 14 | 16 | 18 | 20 | 22 | 24 | 26 | 28 | 30 |
| 4 | | 3 | 6 | 9 | 12 | 15 | 18 | 21 | 24 | 27 | 30 | 33 | 36 | 39 | 42 | 45 |
| 5 | | 4 | 8 | 12 | 16 | 20 | 24 | 28 | 32 | 36 | 40 | 44 | 48 | 52 | 56 | 60 |
| 6 | | 5 | 10 | 15 | 20 | 25 | 30 | 35 | 40 | 45 | 50 | 55 | 60 | 65 | 70 | 75 |
| 7 | | 6 | 12 | 18 | 24 | 30 | 36 | 42 | 48 | 54 | 60 | 66 | 72 | 78 | 84 | 90 |
| 8 | | 7 | 14 | 21 | 28 | 35 | 42 | 49 | 56 | 63 | 70 | 77 | 84 | 91 | 98 | 105 |
| 9 | | 8 | 16 | 24 | 32 | 40 | 48 | 56 | 64 | 72 | 80 | 88 | 96 | 104 | 112 | 120 |
| 10 | | 9 | 18 | 27 | 36 | 45 | 54 | 63 | 72 | 81 | 90 | 99 | 108 | 117 | 126 | 135 |
| 11 | | 10 | 20 | 30 | 40 | 50 | 60 | 70 | 80 | 90 | 100 | 110 | 120 | 130 | 140 | 150 |
| 12 | | 11 | 22 | 33 | 44 | 55 | 66 | 77 | 88 | 99 | 110 | 121 | 132 | 143 | 154 | 165 |
| 13 | | 12 | 24 | 36 | 48 | 60 | 72 | 84 | 96 | 108 | 120 | 132 | 144 | 156 | 168 | 180 |
| 14 | | 13 | 26 | 39 | 52 | 65 | 78 | 91 | 104 | 117 | 130 | 143 | 156 | 169 | 182 | 195 |
| 15 | | 14 | 28 | 42 | 56 | 70 | 84 | 98 | 112 | 126 | 140 | 154 | 168 | 182 | 196 | 210 |
| 16 | | 15 | 30 | 45 | 60 | 75 | 90 | 105 | 120 | 135 | 150 | 165 | 180 | 195 | 210 | 225 |

The next section of the macro selects columns B through P and sets their width to 4. The statement `Columns("B:P").Select` is a property of the worksheet, so prefix it with the `ws.` identifier and add the parenthesis to make it a Python function call. In the next statement, `Selection` is a property of `excel`, so prefix it as such. The translated statements are shown below.

```
>>> ws.Columns("B:P").Select()
True
>>> excel.Selection.ColumnWidth = 4
>>> |
```

Ln: 42 Col: 4

The Excel spreadsheet is now complete, the multiplication table has been expanded to 15×15 and the columns have been resized to 4. At this point, translation of the macro is complete, but the modified file has to be saved. To write the file and quit Excel, use the `SaveAs` and `Quit` methods as shown below.

```
>>> wb.SaveAs('NewMultiplicationTable.xlsx')
>>> excel.Application.Quit()
>>> |
```

Ln: 44 Col: 4

For your reference, here is the [complete Python script](#).

```
#
# make15x15.py
# Expand an existing 10x10 multiplication table and resize columns
#
import win32com.client as win32
excel = win32.gencache.EnsureDispatch('Excel.Application')
wb = excel.Workbooks.Open('MultiplicationTable.xlsx')
excel.Visible = True
ws = wb.Worksheets('Sheet1')
ws.Range("B11:K11").Select()
excel.Selection.AutoFill(ws.Range("B11:K16"), win32.constants.xlFillDefault)
ws.Range("K2:K16").Select()
excel.Selection.AutoFill(ws.Range("K2:P16"), win32.constants.xlFillDefault)
ws.Columns("B:P").Select()
excel.Selection.ColumnWidth = 4
wb.SaveAs('NewMultiplicationTable.xlsx')
excel.Application.Quit()
```

If this is the first time you've ported an Excel macro from VB to Python, congratulations! Please note that in this example, things are kept simple and there is absolutely no error checking or exception handling used here. Normally you would need to provide at least a minimal level of error checking and exception handling in your script so that common errors (missing input file, can't invoke Excel, etc) are caught and handled nicely. Also, this example was developed using Excel 2007, but you can run this code verbatim on earlier versions of Excel if you change the `.xlsx` file extension to `.xls` throughout the script.

Some Porting Guidelines

- Prefix the `Range().Select` statements with the variable name pointing to the worksheet (`ws` in this example)
- Append `()` to any functions
- Prefix the `Selection` statements with the variable name for the Excel spreadsheet (`excel` in this example)
- Prefix the `Columns` statements with the variable name for the worksheet (`ws` in this example)

Porting Reference Table for this example

Note that I didn't capture the `Workbooks.Open()` or `Workbooks.SaveAs` lines in the VB script, it's left as an exercise for the reader to research those commands.

| VB | Python |
|---|--|
| | <code>import win32com.client as win32</code> |
| | <code>excel = win32.gencache.EnsureDispatch('Excel.Application')</code> |
| | <code>wb = excel.Workbooks.Open('MultiplicationTable.xlsx')</code> |
| | <code>wb = excel.Workbooks.Open('MultiplicationTable.xlsx')</code> |
| | <code>excel.Visible = True</code> |
| | <code>ws = wb.Worksheets('Sheet1')</code> |
| <code>Range("B11:K11").Select</code> | <code>ws.Range("B11:K11").Select()</code> |
| <code>Range("B11:K11").Select</code> | <code>ws.Range("B11:K11").Select()</code> |
| <code>Selection.AutoFill Destination:=Range("B11: K16"), Type:=xlFillDefault</code> | <code>excel.Selection.AutoFill(ws.Range("B11:K16"),win32.constants.xlFillDefault)</code> |
| <code>Range("K2:K16").Select</code> | <code>ws.Range("K2:K16").Select()</code> |
| <code>Selection.AutoFill Destination:=Range("K2:P 16"), Type:=xlFillDefault</code> | <code>excel.Selection.AutoFill(ws.Range("K2:P16"),win32.constants.xlFillDefault)</code> |
| <code>Range("K2:P16").Select</code> | <code>ws.Columns("B:P").Select()</code> |
| <code>Columns("B:P").Select</code> | <code>ws.Columns("B:P").Select()</code> |
| <code>Selection.ColumnWidth = 4</code> | <code>excel.Selection.ColumnWidth = 4</code> |
| | <code>excel.Application.Quit()</code> |

Prerequisites

Python (refer to <http://www.python.org>)

Win32 Python module (refer to <http://sourceforge.net/projects/pywin32>)

Microsoft Excel (refer to <http://office.microsoft.com/excel>)

Source Files and Scripts

Source for the program `make15x15.py` and data text file are available at <http://github.com/pythonexcels/examples>

That's all for now, thanks — Dan

Mapping Excel VB Macros to Python Revisited

Posted on [October 20, 2009](#)

The [last post](#) introduced a technique for recording a Visual Basic macro within Excel and migrating it to Python. This exercise will build on those techniques while leveraging Python for more of the work.

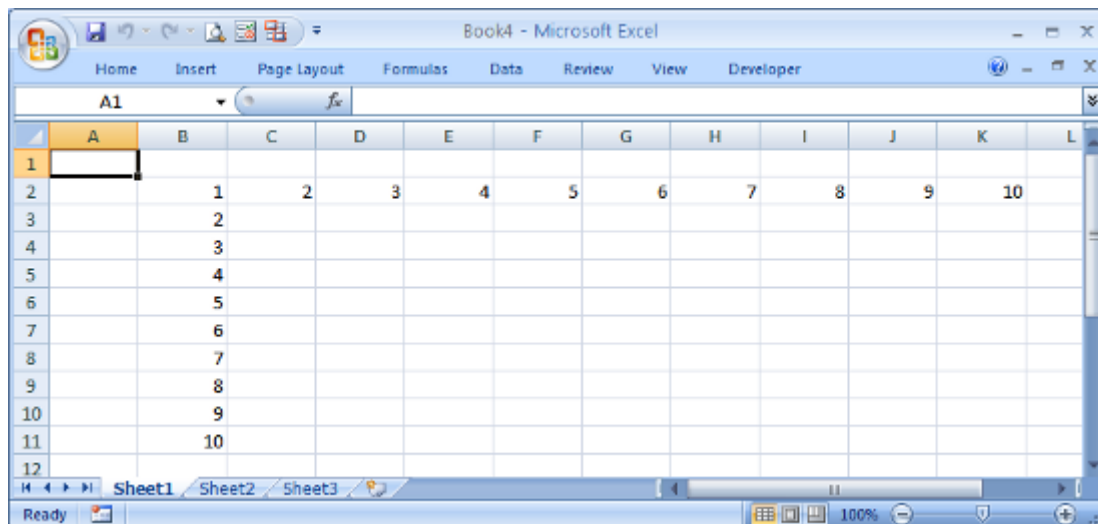
This example creates two tables from scratch – a simple multiplication table and a table of random numbers – and applies conditional formatting to the numbers using some of the new features in Excel 2007 (unfortunately this exercise won't be compatible with older versions of Excel). Begin by starting the Python IDLE interface. Next, start Excel as you've done in the previous exercises. For this exercise, add a workbook using the `Workbooks.Add()` construct, and set the `ws` variable to point to the first worksheet in the workbook.

```
>>> import win32com.client as win32
>>> excel = win32.gencache.EnsureDispatch('Excel.Application')
>>> excel.Visible = True
>>> wb = excel.Workbooks.Add()
>>> ws = wb.Worksheets('Sheet1')
>>> |
```

After typing these command in IDLE, you'll see the Excel window that contains an empty spreadsheet. To build the multiplication table, use Python to populate the column and row headers. There are a number of ways to do this, for this exercise you'll pass an list of column header and row header values to Excel. A row of data is defined by using the `ws.Range().Value` statement with a list or tuple on right hand side of the equals sign. Rather than explicitly defining the list as `[1,2,3,4,5,6,7,8,9,10]`, you can use a functional programming statement containing a `range()` statement to populate the values: `[i for i in range(1,11)]`. The complete statement is `ws.Range("B2:K2").Value = [i for i in range(1,11)]`. Defining a single column of data is a bit trickier, you must define a list of single element lists or tuples. One way to do this is to use Python's `zip()` function to transpose the flat list into a list of tuples. The complete statement is `ws.Range("B2:B11").Value = zip([i for i in range(1,11)])`. The statements for completing the column and row headers are shown below.

```
>>> ws.Range("B2:K2").Value = [i for i in range(1,11)]
>>> ws.Range("B2:B11").Value = zip([i for i in range(1,11)])
>>> |
```

At this point the column and row headers will appear in the Excel spreadsheet.



To define the product values for each cell in the table, create a formula to multiply the column and row header for a single cell, then used Excel to autofill the remaining cells. Looking at the spreadsheet, the product for cell C3 is cell B3 multiplied by cell C2, or 2 times 2 which equals 4. In terms of Excel, the formula is `=B3*C2`. To use Excel's autofill capability, you need to anchor the row and column in the formula by preceding it with the `$` character. In other words, the formula you want to use is `=$B3*$C$2`. Once that formula is entered, the expansion to fill the remaining cells is done in two steps. First, programmatically select the cell and drag it fill the row. Next, select the newly autofilled row and drag the new row down to fill in the remaining rows. Since this was demonstrated in the [last post](#), please refer to that post if you need more information. The equivalent Python code to implement the autofill is shown below.

```
>>> ws.Range("C3").Formula = "=$B3*$C$2"
>>> ws.Range("C3:C3").Select()
True
>>> excel.Selection.AutoFill(ws.Range("C3:K3"), win32.constants.xlFillDefault)
True
>>> ws.Range("C3:K3").Select()
True
>>> excel.Selection.AutoFill(ws.Range("C3:K11"), win32.constants.xlFillDefault)
True
>>> |
```

The spreadsheet will now contain the complete multiplication table.

| | A | B | C | D | E | F | G | H | I | J | K | L |
|----|---|----|----|----|----|----|----|----|----|----|-----|---|
| 1 | | | | | | | | | | | | |
| 2 | | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | |
| 3 | | 2 | 4 | 6 | 8 | 10 | 12 | 14 | 16 | 18 | 20 | |
| 4 | | 3 | 6 | 9 | 12 | 15 | 18 | 21 | 24 | 27 | 30 | |
| 5 | | 4 | 8 | 12 | 16 | 20 | 24 | 28 | 32 | 36 | 40 | |
| 6 | | 5 | 10 | 15 | 20 | 25 | 30 | 35 | 40 | 45 | 50 | |
| 7 | | 6 | 12 | 18 | 24 | 30 | 36 | 42 | 48 | 54 | 60 | |
| 8 | | 7 | 14 | 21 | 28 | 35 | 42 | 49 | 56 | 63 | 70 | |
| 9 | | 8 | 16 | 24 | 32 | 40 | 48 | 56 | 64 | 72 | 80 | |
| 10 | | 9 | 18 | 27 | 36 | 45 | 54 | 63 | 72 | 81 | 90 | |
| 11 | | 10 | 20 | 30 | 40 | 50 | 60 | 70 | 80 | 90 | 100 | |
| 12 | | | | | | | | | | | | |
| 13 | | | | | | | | | | | | |

To help illustrate conditional formatting, create another table of random integers between 1 and 100. Excel's `RAND()` function will generate a random number between 0 and 1, the formula we want is `=INT(RAND()*100)`. The `ws.Range().Formula` construct can be used to fill a range with the same identical formula.

The Excel spreadsheet should now contain both the multiplication and random number tables.

Now that the data is ready, conditional formatting can be applied. Even though you invoked Excel from Python, you still can manipulate the spreadsheet using the Excel interface, and even record macros. You need to record a macro like you did in the [last post](#) in order to capture the VB commands, so click on Record Macro in the Developer tab, then click OK in the popup dialog.

Select all the cells in the range `B2:K22`. In the Home tab, select Conditional Formatting->Color Scales->Red-Yellow-Blue Color Scale.

The spreadsheet should now show a color background for each of the selected cells containing a value. Now select cell A1, then stop the macro by clicking Stop Recording in the Developer tab.

Your spreadsheet will now have conditional formatting applied and will look something like this, with cells containing numbers near 100 colored in Red, cells with a value of 50 in Yellow, and cells with a value of 1 in Blue, with a shade of these colors for values in between:

As an aside, you can update the random numbers in the lower table by hitting the F9 key to force a spreadsheet recalculation.

To continue, open the macro just created by selecting Macros from the Developer tab, select the name of the macro you just captured and click Edit. The macro should look something like this:

```
sSub Macro1()  
'  
' Macro1 Macro
```

```

Range("B2:K22").Select
Selection.FormatConditions.AddColorScale ColorScaleType:=3
Selection.FormatConditions(Selection.FormatConditions.Count).SetFirstPriority
Selection.FormatConditions(1).ColorScaleCriteria(1).Type = _
xlConditionValueLowestValue
With Selection.FormatConditions(1).ColorScaleCriteria(1).FormatColor
.Color = 13011546
.TintAndShade = 0
End With
Selection.FormatConditions(1).ColorScaleCriteria(2).Type = _
xlConditionValuePercentile
Selection.FormatConditions(1).ColorScaleCriteria(2).Value = 50
With Selection.FormatConditions(1).ColorScaleCriteria(2).FormatColor
.Color = 8711167
.TintAndShade = 0
End With
Selection.FormatConditions(1).ColorScaleCriteria(3).Type = _
xlConditionValueHighestValue
With Selection.FormatConditions(1).ColorScaleCriteria(3).FormatColor
.Color = 7039480
.TintAndShade = 0
End With
Range("A1").Select
End Sub

```



[view raw Macro1](#) hosted with [by GitHub](#)

Though the macro contains some very long method names, plus some `With` statements, the porting will be very straightforward. Here are some guidelines to keep in mind while migrating this code to Python.

- `Selection` is preceded by `excel.`

Remember that `Selection` is a method at the Excel Application level, you need to precede it with `excel.` in this example.

- `Range` is preceded by `ws.`

`Range` is a method at the Worksheet level, which is defined earlier as `ws.` in this example.

- Function calls require `()` in Python

Unlike VB, any function calls must be followed by `()` in Python.

- `With` statements must be expanded

The three `With` blocks in this macro need to be expanded, which can be done with temporary variables or by copying the statement following the `With` keyword. For example, the first `With` block:

```
With Selection.FormatConditions(1).ColorScaleCriteria(1).FormatColor
    .Color = 13011546
    .TintAndShade = 0
End With
```

can be written in Python as

```
excel.Selection.FormatConditions(1).ColorScaleCriteria(1).FormatColor.Color
= 13011546
excel.Selection.FormatConditions(1).ColorScaleCriteria(1).FormatColor.TintA
ndShade = 0
```

or by using a temporary variable as

```
x = excel.Selection.FormatConditions(1).ColorScaleCriteria(1).FormatColor
x.Color = 13011546
x.FormatColor.TintAndShade = 0
```

Temporary variables were created to make the script more concise. In particular, the statement `[csc1, csc2, csc3] = [excel.Selection.FormatConditions(1).ColorScaleCriteria(n) for n in range(1,4)]` was used to create three temporary variables for the three `ColorScaleCriteria` methods. The equivalent Python text representing the macro is shown here:

To save the spreadsheet and close Excel, use the `SaveAs` and `Quit` methods as shown below.

Here is the [complete conditionalformatting.py script](#). The line `excel.Visible = True` has been commented out. Unless you are developing the script, you typically want Excel to run invisibly in the background.

```
#
# conditionalformatting.py
# Create two tables and apply Conditional Formatting
#
import win32com.client as win32
excel = win32.gencache.EnsureDispatch('Excel.Application')
#excel.Visible = True
wb = excel.Workbooks.Add()
```



```

ws = wb.Worksheets('Sheet1')
ws.Range("B2:K2").Value = [i for i in range(1,11)]
ws.Range("B2:B11").Value = zip([i for i in range(1,11)])
ws.Range("C3").Formula = "=$B3*C$2"
ws.Range("C3:C3").Select()
excel.Selection.AutoFill(ws.Range("C3:K3"),win32.constants.xlFillDefault)
ws.Range("C3:K3").Select()
excel.Selection.AutoFill(ws.Range("C3:K11"),win32.constants.xlFillDefault)
ws.Range("B13:K22").Formula = " =INT(RAND()*100)"
ws.Range("B2:K22").Select()
excel.Selection.FormatConditions.AddColorScale(ColorScaleType = 3)
excel.Selection.FormatConditions(excel.Selection.FormatConditions.Count).SetFirstPriority()
[csc1,csc2,csc3] = [excel.Selection.FormatConditions(1).ColorScaleCriteria(n) for n in
range(1,4)]
csc1.Type = win32.constants.xlConditionValueLowestValue
csc1.FormatColor.Color = 13011546
csc1.FormatColor.TintAndShade = 0
csc2.Type = win32.constants.xlConditionValuePercentile
csc2.Value = 50
csc2.FormatColor.Color = 8711167
csc2.FormatColor.TintAndShade = 0
csc3.Type = win32.constants.xlConditionValueHighestValue
csc3.FormatColor.Color = 7039480
csc3.FormatColor.TintAndShade = 0
ws.Range("A1").Select()
wb.SaveAs('ConditionalFormatting.xlsx')
excel.Application.Quit()

```



[view raw conditionalformatting.py](#) hosted with [by GitHub](#)

Prerequisites

Python (refer to <http://www.python.org>)

Win32 Python module (refer to <http://sourceforge.net/projects/pywin32>)

Microsoft Excel (refer to <http://office.microsoft.com/excel>)

Source Files and Scripts

Source for the program conditionalformatting.py script is available at

<http://github.com/pythonexcels/examples>

That's all for now, thanks — Dan