# PyJEM: A How To Guide.

D. R. G. Mitchell,

www.dmscripting.com,

Oct 2022

## What is PyJEM?

PyJEM is a Python software interface which enables external Python scripts to interact and control a PyJEM-equipped JEOL microscope. Microscope models which are compatible with PyJEM include 1400+, 2100, ARM, F200, 2800. However, like me you may have an older model of one of these microscopes (my ARM is 7 years old), and it may or may not have PyJEM functionality present. You can check if you have it, by reading on below. It is not clear to me what might be involved in getting an older microscope to be compatible with PyJEM. This would probably involve an update to the latest version of TEMCentre control software. I'll investigate this on my ARM.

## Why use PyJEM?

PyJEM enables the creation of bespoke experimental capabilities, such as novel microscope configurations providing improved functionality. The microscope control functionality provided by PyJEM is far in excess of what is available in DigitalMicrograph script (DM Script). In DM Script, microscope commands are necessarily a limited subset of what is available, due to the need to make them vendor-agnostic. Old time JEOL users will remember the JEOL command language on serial communication microscopes like the 2000/3000FX, 2010/3010 etc. With that language you could control the entire microscope. That capability was largely lost when JEOL moved to Ethernet communication, and scripters were then restricted to the much smaller set of functionality provided in DM Script. So for anyone interested in controlling their microscope externally, PyJEM is big news. It is also interesting to note that the very latest version of DigitalMicrograph supports Python. If DM Script could access PyJEM's functionality, then that would be a powerful synergy.

## What software makes up PyJEM?

PyJEM consists of two main parts: Anaconda3 and PyJEM itself. Anaconda3 is a package providing a suite of tools with which to develop Python scripts. This is just a regular Python development environment and is not something unique to JEOL microscopes. The important parts of Anaconda3 are:

1. Spyder: This is a development environment in which Python scripts can be written, run and tested. Within Spyder is a Console which displays the output from the script and any error messages.
2. iPython Console: A standalone command line which allows you to enter Python commands and run Python scripts.

3. Anaconda Prompt: A command line – like the command line in Windows or the Terminal in MacOS. This allows you to run commands to update/start/stop/delete packages etc. Be very careful about updating/changing software attached to a microscope. If you have a working system, leave it be. The Anaconda3 eco-system seems to be very particular about which components work with which. If you update one part, you might find others no longer work with it.

For novice Python scripters like me, you will likely spend nearly all of your time in Spyder.

The second part of the system, is the PyJEM installation itself. This provides modules of microscope functionality which can then be accessed using Python. Copies of PyJEM installer files get stored in C:\Program Files\JEOL\PyJEM_x64, and installed into Anaconda3 by the PyJEM installer program.

Getting Started.

Your PyJEM-equipped microscope should already have Anaconda3 installed with copies of the PyJEM installation files installed in C:/Program Files/JEOL/PyJEM_x64 and relevant libraries installed into Anaconda3. If so, you can select Spyder from the Anaconda3 option in the Windows menu and begin bashing out a Python script. However, most microscopes are extremely busy, and are best used for microscopy, rather than code development. It is much better to develop scripts offline on a system not connected to a microscope. Only for the final testing and polishing should the script be moved to the microscope. So the first step is to set up an offline instance of PyJEM, where you can do all the preliminary coding.

Installing Software:

When setting up offline instances of PyJEM (all on Windows 10 systems), I have followed the detailed procedure described in the JEOL PyJEM insert found at the back of the microscope manual, and found it to work as advertised. I only encountered one instance where it would not work. This was on a very new Dell work laptop. The installation procedure completed successfully, but Spyder would always quit at launch. On all other computers on which I tested this, none of which were connected to my work domain, everything worked as advertised. On the troublesome work laptop I tried installing PyJEM on top of the most recent version of Anaconda3 (5.0). (The JEOL-supplied version (4.0) being 2 years old). The latest version of Anaconda3 would install and run fine, but PyJEM wouldn't play with it. For this reason, if you have a setup of Anaconda3/PyJEM which works – do not update it.

The PyJEM software can be found on the TEM External COM Interface CD which is part of the set of JEOL software disks which came with the microscope. Look for the folder called PyJEM Installer. At microscope installation, the JEOL engineer should have set up PyJEM on the microscope control PC. If you click on the Windows icon (bottom left of Windows screen), look for Anaconda3 in the popup menu. If it is present, then it is likely that PyJEM is installed. To confirm it is installed look in

C:/Program Files/JEOL for a folder called PyJEM_x64. If this folder is present, PyJEM is installed and you are ready to go. If not consult with JEOL.

For all script development you should install Anaconda3 and PyJEM on an offline computer. Here you will do nearly all of your script writing. You do not need to be at the microscope in order to do preliminary development and testing of scripts, because JEOL provide two sets of modules within PyJEM – one set is called PyJEM and the other is called PyJEM.offline. The former is used at the microscope and provides real microscope responses, while the latter is used in offline scripts during development and provides a simulated set of responses. For example, if you use the PyJEM.offline module and run the command GetMicroscopeMag(), the offline module will return a sensible (but unchanging) value of 200x. With this response, you can confirm that your script works. However, you will eventually need to test it at the microscope, particularly if your script is looking to change magnification, or do certain things at certain magnifications.

Where is the Documentation and Example Code?

Now that you have a shiny new setup of Anaconda3 and PyJEM on your office PC, you are all ready to start coding. However, what you need in order to do that is a detailed set of documentation, which explains how to write scripts, what commands are available, what is the syntax of each command – with example code, and a repository of actual scripts which provide succinct code which is well annotated, to explain how to do certain things. If anybody knows where to find all of those things, please be sure to let me know. The problem with this type of functionality, is that it is really nice to have, but it doesn't directly contribute to the vendor's bottom line. So writing all of the above is an extremely low priority. I am not being harsh on JEOL- it is just a commercial reality. During my first decade of writing DM Script, the documentation and support from Gatan was similarly missing in action. Ten years on and all the above is now on line, easily accessible and of very high quality and utility. I have no doubt that PyJEM support will gradually improve as documentation catches up with code. But as with my experiences with DM Script, the (personal) solution is to develop and share those resources, in the hope that other, like-minded souls, both inside and outside of JEOL, will do likewise. At the time of writing the resources which I have found to be useful include:

1. GitHub PyJEM repository: GitHub - PyJEM/PyJEM. Curently this is very sparse, but hopefully it will grow if users contribute. I will post all my PyJEM output on my website (www.dmscripting.com).
2. HTML pages listing all the functions available. These can be found inside the PyJEM folder, which on my computer lives in : C:\Program Files\JEOL\PyJEM_x64\library\PyJEM-1.0.2.1143.zip\PyJEM-1.0.2.1143\PyJEM\doc\interface. Inside the PyJEM_x64 folder is a library file. This stores various zip archives – the most recent of which the PyJEM installer unzips and installs (in Anaconda3). You can navigate down the above (very long) path and you will see HTML pages inside the interface folder, which describe all the commands available. I copied the PyJEM_1.0.2.1143.zip folder and extracted a copy onto my desktop. This made it a lot more convenient to access.

3. If you extracted the above folder to your desktop, in it you will find some Sample code at C:\Users\Username\Desktop\PyJEM-1.0.2.1143\PyJEM\samples (where Username is the name of your account on the computer.)

Writing your first Python script

I am fairly new to Python. Since it is a C-##-like object oriented language, some of the basic ideas and concepts are similar to DM Script with which I am familiar. However, the implementation and command syntax of Python are quite different, so it took me a couple of weeks to get my head into it. As with all things, once you understand it, it is not hard. However, getting help form online forums etc is a real minefield, because there are different flavours of Python, so the subtle differences get you. There is much about Python I dislike. Unlike DM Script – capitalisation matters, as does white space. Functions and loops need to be indented and a misplaced capital or an extra space can be very hard to spot. The code will not run if there is a problem. The Spyder Console reports errors, but the error reporting of limited utility. Syntax errors are common enough, but at least DM Script suggests some functions/syntax which you may have been trying to enter. There is a huge Python community out there. I think a lot of web programming is done in Python. As such there are lots of people for whom Python is their day job. In the DM Script community most of us are not professions and do this in our spare time, and so we remember what it is like to be a confused novice. The Python online forums, often immediately get lost in the weeds of detail, without helping novices join the dots. To my mind, the activation energy in getting up to speed with Python is greater than DM Script.

Writing Scripts.

Hello World Script.

Open Spyder.

In the script area of Spyder type the following:

```
# Lines beginning with a # are comments.

# Script starts here

Print(“Hello World”) # this is the only line of the script which does anything

# Script ends here
```

This command will print Hello World in the Console. The Console should be shown at either the bottom or the bottom right of your Spyder window. If you copy and paste the above code into

Spyder, you may have to edit the quotes in print(“ Hello World”), as Spyder expects straight quotes and curly quotes will throw an error.

Every time you run a script in Spyder it will first save the code. This is in contrast to DM Script – which is only saved when you save it. If you haven’t saved your Python ‘Hello World’ script before running it, you will be prompted to save it first – do so. Save it as Hello_World. The problem with this ‘save then run’ approach, is that if you make a large change which causes a catastrophic failure of your script, you cannot go back to the previous version, unless you saved it earlier.

To run the script click on the green chevron in the toolbar  - top left in Spyder.

Each time you run the script you will see   Hello World   appear in the Console.

Sourcing the Microscope Magnification:

Create a new script window in Spyder: File/New File

Begin by importing the PyJEM module you will need. Here we will be running this offline, so we will import from the PyJEM.offline module. If you are running this at the microscope then you will instead import from the PyJEM module (ie change from PyJEM.offline import TEM3  to from PyJEM import TEM3

```
#Script begins here

#From the PyJEM offline, import the TEM3 module which contains all the functionality we need to access.

from PyJEM.offline import TEM3 # to test this at the microscope remove .offline

# Now create an instance of the EOS3 module. This is a submodule in TEM3 which contains the command to access the
magnification.

my_eos_module=TEM3.EOS3()

# Source the microscope magnification and print the value

mag_val=my_eos_module.GetMagValue()

print("Microscope Mag is = "+str(mag_val)) #unlike DM script any non-string

# variables must be explicity converted to a string (using the str( ) notation)

# prior to concatenating them into a string. In # DM Script this happens automatically.

# Script ends here.
```

When you run the script the output you get is:

```
Microscope Mag is = [200, 'X', 'X200']
```

The mag_val variable is a three item list, containing the numerical value of the mag, the multiplier (X) and the combination of both. You can extract a particular element, in this case the left-most (position 0) with the command:

```
my_mag=mag_val[0]
```

The above script has been edited to include this additional command and is shown below:

```
# Script begins here

#From the PyJEM offline, import the TEM3 module which contains all the functionality we need to access.

from PyJEM.offline import TEM3 # to test this at the microscope remove .offline

# Now create an instance of the EOS3 module. This is a submodule in TEM3 which contains the command to access the magnification.

my_eos_module=TEM3.EOS3()

# Source the microscope magnification and print the value

mag_val=my_eos_module.GetMagValue()

my_mag=mag_val[0]  # get the first item in the list – which is position 0

print("Microscope Mag is = "+str(my_mag))

# Script ends here.
```

## Troubleshooting scripts.

### Errors

As you enter lines of code and run them Spyder, you will encounter lots of errors. The error stopping execution will be reported in the Console window of Spyder. Often times these will be syntax errors. For example if a line of your code says Print(“Microscope Mag = “+str(my_mag)) it will throw an error saying ‘name error Print is not defined’. This is because the print command must be print() and not Print(). Another common error is ‘EOL while scanning string literal’. This occurs because you have opened a set of quotes but you have not closed them eg print(“Hello) . Indentation is really important in Python. If you define a loop, such as a For loop, or you define a function, everything which follows which belongs to that loop or function, must be indented by one tab. If you get it wrong, you will get ‘unexpected indent’ errors.

Crashed Console.

Sometimes your script skittles the Console. It just appears to stop, does nothing or your dialog freezes up. When a script is actively running, there is a little square in the top right corner of the Console which turns from grey to red. For scripts which produce dialogs or run background threads, these will run constantly. However, if you have a script which should launch, do its thing, then finish, but the square stays red, then you have a crash. The simplest thing to do is to close the Console. You can see in the top left corner of the Console is a tab with the Console name eg Console 2/A and next to it is a red checkbox. Click on that to close the Console. You will need to open a new Console in Spyder by selecting Consoles/Open a new iPython Console. You will not be able to run your script until you have a Console open which it can use.

One little gotcha I have become aware of is that the Console has a memory. If I run script A in Console B and in so doing Script A imports various modules and functionality and runs fine – all is well. If I next run script C in Console B, there is a memory of what modules/functionality I imported. If Script C relies on functionality which it should have imported itself, but its gormless coder forgot to include, then Script C may still run OK – because what it needs, Console B already has on board. However, fast forward to next day, when I continue working on Script C, this time using a rebooted Spyder and a new Console D, lo and behold!, the script which worked perfectly the day before no longer works. This can be confusing. A good rule of thumb is whenever your finish using one script, close the Console, then open a fresh Console and then carry out development of your new script in a fresh Console. That way, anything you have omitted to include will be obvious from the get-go.

Running scripts.

The above examples show you how to write and run scripts in Spyder and monitor the output in the Console. There are other ways of running scripts.

Running scripts by clicking on them.

When you save your scripts they have the extension .py. You should set these files to be opened by Python by default. If they have a generic grey file icon, then this has not been set. Find one such .py file, right click on it and select Open With making sure the 'Always Open With' option is also checked. Then search for the Python application (not pythonw). This lives in C:\Anaconda3 and click on it to select it as the app with which to open .py files. Thereafter, you can run these scripts simply by clicking on them. They will launch automatically in Python and run. The problem with running a script, such as the one above to get magnification, is that the script will launch and display a Console window, output the mag to the Console window, and then immediately close ie you will not see what the script is doing before it has closed itself. The example code below shows that this method does work, for some types of scripts – such as those producing dialogs. Dialogs are run in an endless loop and so persist on the screen, they don't just open, do their thing then close. Dialogs are an advanced topic, and I won't cover them here, but for now, just run the script below to convince yourself that a saved python script can be run by double clicking it. Don’t worry about the code, it is just designed to display a dialog with a button. This stays open until the user clicks on it:

```
#Script begins here

from tkinter import *

from tkinter import messagebox


ws = Tk()

ws.title('Script Test')

ws.geometry('200x200')


def button_response():

    messagebox.showinfo('Button Response', 'Closing the Window.')

    ws.destroy()



Button(ws, text='Click Me', command=button_response).pack(pady=50)

ws.mainloop()

# Script ends here
```

## Running scripts within iPython Console

If you are familiar with the command line in Windows/Linux or the Terminal in MacOS, then this will be straightforward. If you have assiduously avoided the command line then it is a bit more work than double clicking, but not much.

Firstly, when you save your scripts you should avoid spaces in the name. Use underscores for spaces – but not hyphens ie save the Hello World script as hello_world.py. When using the command line spaces cannot be included in file paths – unless closing quotes are used. Personally I find it easier to always use an underscore. If you open iPython Console from the Anaconda3 menu, and navigate to the folder where the script is stored, you can run a script. For example to run a script called hello_world.py use the following command:

```
run hello_world.py
```

The script will run and the output will appear in the Console.

If you script was called   hello world.py    ie it contained a space. Then you have to enclose the file name in quotes ie

run “hello world”

Note my word processor uses curly quotes but in Python you must use straight quotes.

This works fine if you have navigated to the folder where your script lives. However, you can run scripts which are in other folders by typing run, then space, then click and drag the script you want to run onto the Console window. This will complete the full path to the file and you can then then hit return to run it.

Navigating in iPython is not too difficult. The two commands you will need to know are   cd  to change directories and   ls   to list the contents of the directory.

cd \Users\username\Desktop   (where username is your user account name on the computer) will take you to your desktop.

cd .. will navigate one step up the directory tree.

cd ~ will navigate to your home directory ie \Users\username

cd / will navigate to the root of the c directory ie c:\

If you type cd, then a space, and then click and drag your target folder into the iPython Console window, the path of that folder will complete the rest of the line for you – very useful for really long paths.

ls  will list the contents of whichever directory you are currently in.

Use the above commands in iPython Console to navigate to the directory which contains your python file. Then run it by typing    run filename.py   where filename is the name of your script.

Next Steps.

What you do with Python and PyJEM is limited only by your imagination. I am currently finalising two scripts. One will break out aperture control to the keyboard arrow keys. The software interface on the F200 (I do not have (hardware) aperture control buttons on my F200), redefines the word exasperating. I am also currently testing a script to configure the microscope for Ronchigram and Beam Shower modes. These have the potential to speed up my microscopy and reduce errors. Over the years I have inadvertently carried out a surprising number of beam showers with the Beam Valve closed – my script tells me if I am being a numpty. I have lots more ideas. If you have a good idea for a script which will be widely useful to the JEOL TEM/STEM community, but lack the chops to do it yourself, you are welcome to pitch the idea to me.

I am not a Python expert, and have completed only three Python projects of significance. If you have a general Python-related query, I would suggest that you seek assistance from the many forums out there such as Stack Overflow, which are monitored by experienced Pythonistas. If you have PyJEM-specific queries, you are welcome to ask.