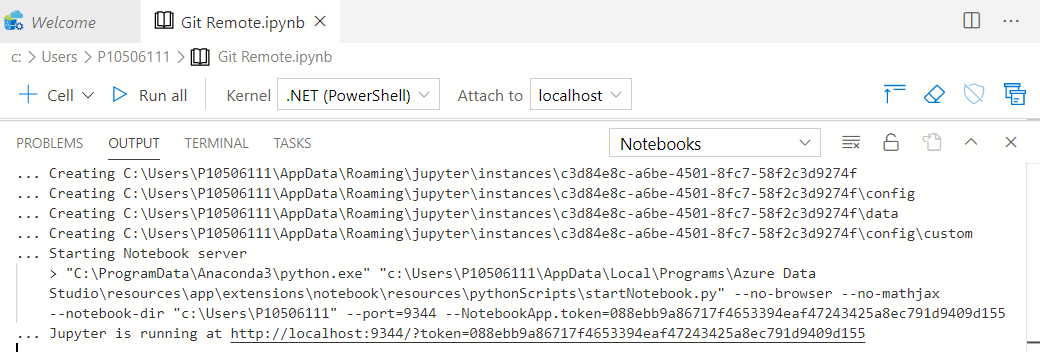
In Part 1 I covered installing a version of Python with the Jupyter server and adding extra Kernels – in particular, .NET Interactive for PowerShell support. After a quick exchange with some friends on twitter it seems I could have explained something better in part 2 about the different ways .NET Interactive works with Azure Data Studio, and visual studio code, which is what I’m going to try to cover here.

The Jupyter server can work in two different ways. My set up of default anaconda installation + .net Interactive from part 1 gives the ability to start the server and access it through a web browser to work with notebooks stored as .ipynb files.

If Azure data studio (or the Python extension in VS Code) is told to open a .ipynb file it starts python to run exactly the same web server specifying the port and access token and then makes API calls to use the server, without a web front end. You can see this in Azure data studio below, and the link at the bottom will open in a browser allowing two kernels on one server to see the same set of files. The two kernels thing matters because there is no interaction between code in one and code in the other.



Getting the Process object for the current PowerShell process with Get-process -id $pid and then checking its parent process, than that process’s parent and so shows that   
Azuredatastudio -runs- cmd -to launch- python -which runs Jupyter and starts a kernel -dotnet -which hosts- dotnet-interactive

So Azure data studio is using Python Jupyter bits except for rendering as HTML/Javascript - because it does its own rendering.

I said in part 2 that the .NET interactive extension for VS code uses its own installation – when Python launches the kernel it runs  
<home>\.dotnet\tools\dotnet-interactive.exe Jupyter <parameters>

Things are different in VS code   
Code runs dotnet tool run <parameters> which runs dotnet <home>\.nuget\…\ Microsoft.DotNet.Interactive.App.dll <parameters>

No python, no Jupyter web server, it goes directly to the Kernel loading files from the .nuget directory instead of the .net one.

I also referred to the difference between Jupyter-generated output and output from the DotNet Interactive extension for VS code (I’m tempted to start abbreviating that to DNIE pronounced like deny). Jupyter creates JSON inside the ipnyb file like this

"outputs": [

{

"output\_type": "stream",

"name": "stdout",

"text": [  
 "Line 1,  
 "Line 2",  
 "Line 3"  
 ]

}

]

It’s also valid to have "text": "A long string with \n for newlines"

*At the moment* the extension for vs code writes like this:

"outputs": [

{ "output\_type": "execute\_result",

"data": { "text/plain": "Line 1" }

},

{ "output\_type": "execute\_result",

"data": { "text/plain": "Line 2" }

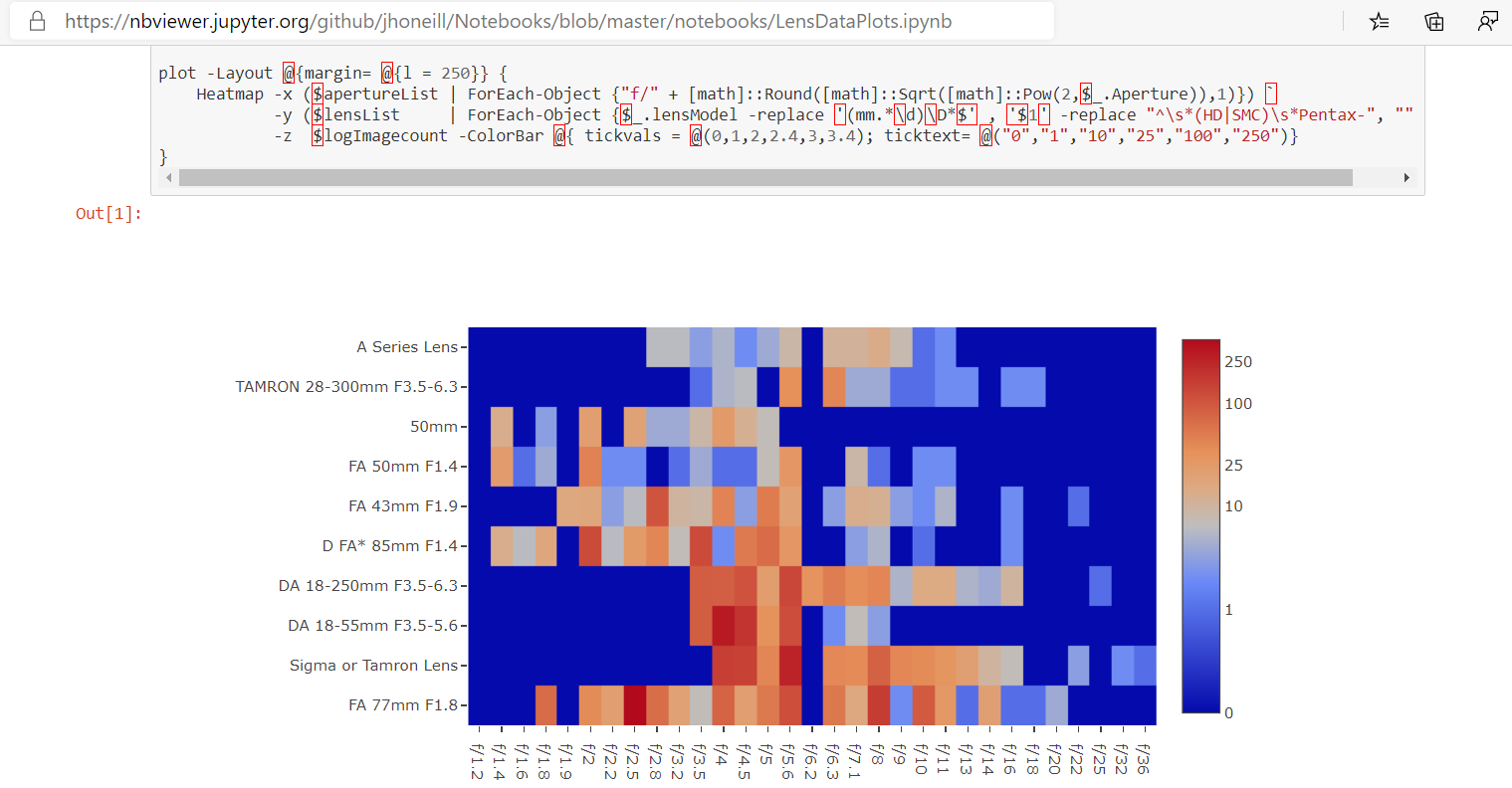
}

]

Both get read everywhere but often the Jupyter format reads more nicely. I would stress *at the moment,* because, to repeat what I said in part 2, **this is a preview and things can change**.

Being JSON internally, lots of things can read and/or write ipynb format. The notebook files has a metadata block which says what language it uses, and a cells block, each cell has a type (code or markdown), source, and – for code cells - outputs. My friend Doug Finke has a PowerShell module to help with that. And there are other useful tools:

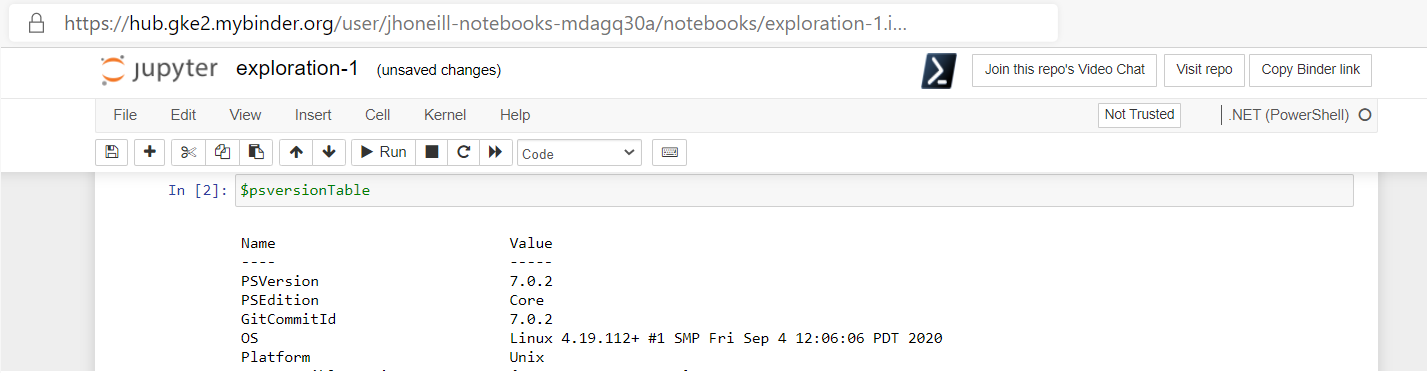
* **GitHub** will display the contents of a .ipynb. file in read-only form -skipping any JavaScript.
* [nbviewer](https://nbviewer.jupyter.org/) seems to have the same templates but doesn’t filter out the Javascript. If I run something on my computer, my results are embedded in the Notebook, and someone I share them with might not have access to my data so being able to show a chart is a consideration



Incidentally, the screenshot above shows a notebook from my git hub repo <https://github.com/jhoneill/Notebooks> - someone asked a question a photography forum which led me to do a heat map of what lenses and setting I use from the lightroom data <https://jhoneill.github.io/powershell/photography/2012/08/09/Lightroom-data.html> I’ve been messing with for years

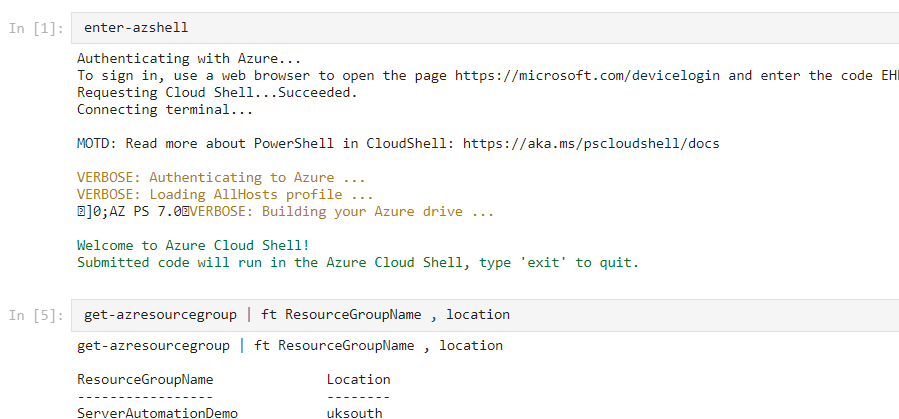
* Binder goes even further. It gives you a docker container running the Jupiter server. It will copy information from a GitHub repo and launch the server. By default, it doesn’t have the additional Kernels from .NET interactive, but you can provide a dockerfile to configure the container – I got mine , and the associated nuget file, from the dotnet interactive repo. <https://github.com/dotnet/interactive/tree/main/samples/my%20binder>

Binder can take a few moments create and launch the container but it’s live, you can change the code and re-run it. The container doesn’t have any access to your machine (so files you need must be in the repo), but it is possible to install modules, or connect to an Azure cloud shell or run other commands which access resources from the internet.



In case you weren’t aware that .NET interactive is cross platform, the screen shot above shows the docker containers for binder run it via Python on Linux. Binder disposes of the container then you leave or if you’re inactive for too long.

I can see running shared notebooks with a service like binder, as a way to do “show your working” configuration or analysis of on-line services – and I’d pop over to print preview in Jupyter to generate a nice view of what has been done. Like this.



As I said being JSON it’s possible to fiddle with the contents of the files, and I built some things based on Doug’s module.

First: Get-NotebookContent returns objects with a type property set to “Code” or “Markdown” and a source property. It only takes a couple of lines either to output the Markdown source as-is or to run the source of a code cell. Because the output may be different types in different commands, and PowerShell will try to format it based on the type of the first one, I force it through Out-Default

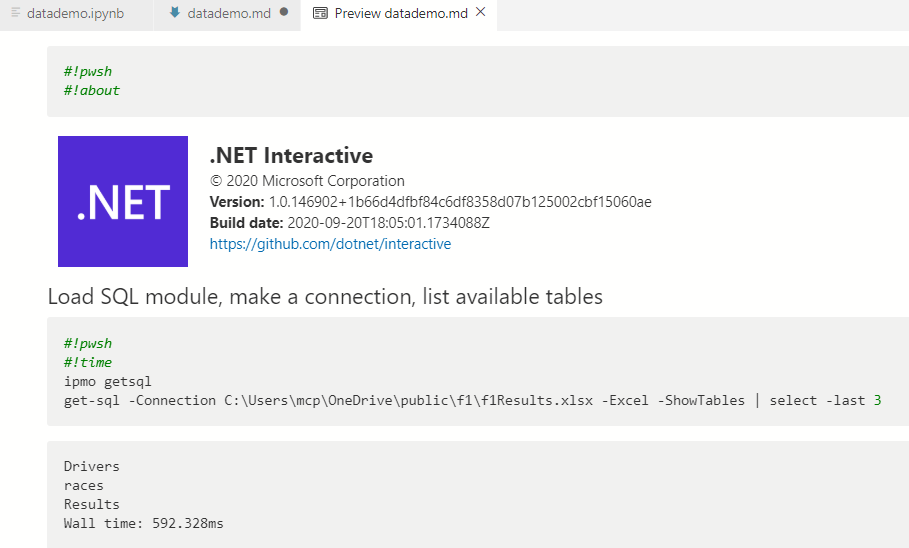
Get-NotebookContent .\datademo.ipynb | foreach-object {

if ($\_.type -eq "code") {Invoke-Expression $\_.Source | out-default }   
 else {Write-Verbose -Verbose $\_.source}

}

I added an option to let me run a command like   
ConvertFrom-NotebookToMarkdown .\datademo.ipynb -Includeoutput

And get this



Doug also created commands which allow a new notebook to be created like this:

New-PSNotebook -NoteBookName .\TestNotebook -IncludeCodeResults {

Add-NotebookMarkdown -markdown "# This is a H1 tag"

Add-NotebookCode -code 'Hello World'

}

-IncludeCodeResults creates a runspace while New-PSNotebook is running, and runs each of the code cells and incorporates their output. I made some tweaks here too – to have -Verbose show the output, to the notebook to be of either a Windows PowerShell one for Azure data studio OR .a .NET interactive ones, I catch a few of the magic commands – removing the redundant #!pwsh and processing #!About and #!time - and I added some aliases to make it a bit more like a domain specific language

notebook {  
 markdown $myMarkdownText   
 code $MyPSScript  
} -DotNetInteractive test.ipynb

I’m not 100% that changing code from pointing to *Visual-Studio code* to *Add-NotebookCode* is a good idea, but for now it gives me an interesting trick: which I’ll come to in a moment. I could combine the last example above with the one to run code from the notebook and make something like this   
notebook {

switch (NotebookContent .\datademo.ipynb) {

{ $\_.Type -eq 'markdown' } { markdown $\_.Source }

{ $\_.Type -eq 'code' } { code $\_.source -Verbose }

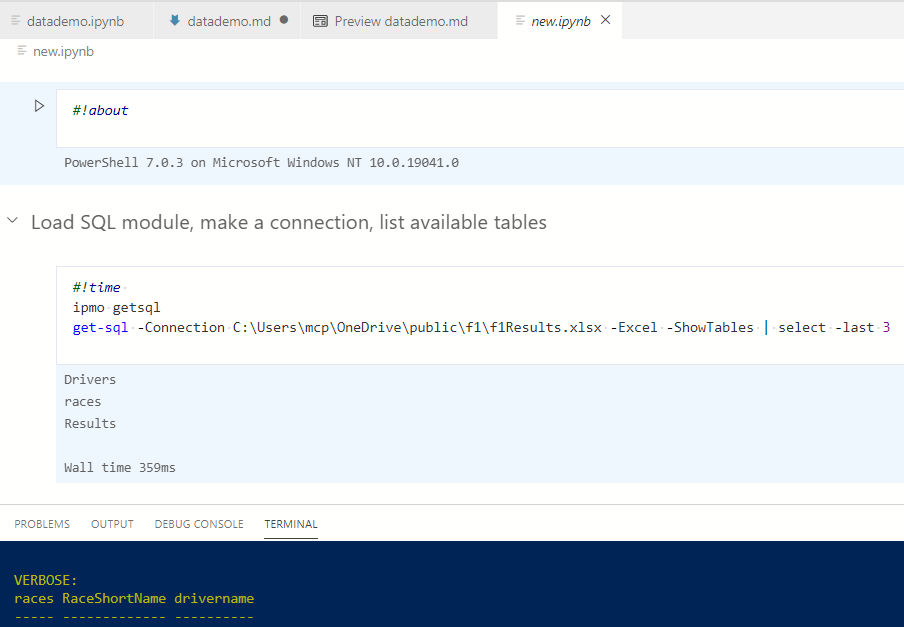
}

} -IncludeCodeResults -DNI dd2.ipynb

The trick I mention is that since I am running (via aliases) markdown if the type field holds “markdown” and code if it holds “code” I could remove the switch statement

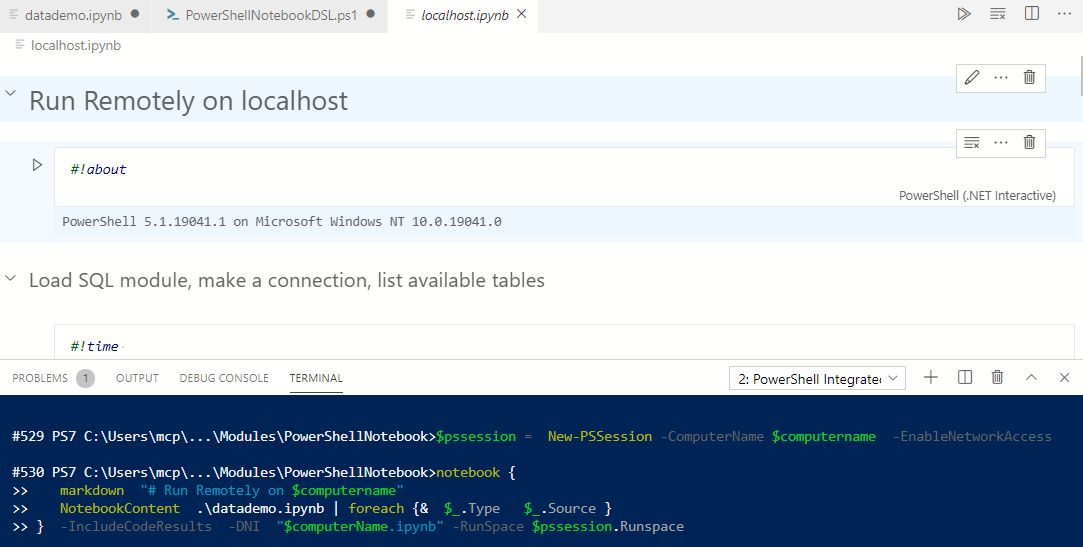
notebook {  
 NotebookContent .\datademo.ipynb | foreach {& $\_.Type $\_.Source -Verbose }  
} -IncludeCodeResults -DNI new.ipynb

That gives me a notebook like the one below.



In the terminal window you can see some of the data which was displayed by -verbose and which will be in the final cell of the notebook. At the top you can see #about and #time have been processed and about shows the version of PowerShell which ran the script.

But while I was writing I had one last idea. Doug uses a runspace. Hmmm. I can connect to remote runspaces… OK I said, let’s add a -runspace option and I got this:



Now… Imagine this was is a block which ran in parallel for many remote machines.