Unpractical Powershell

## Why Read This Book?

When I first started as an IT professional 20+ years ago I loved the GUI (graphical user interface). It was safe, it was friendly, like a fluffy bunny on a warm spring day under the sun. You pressed the button and the computer did something. Pressing buttons and watching status bars…Pressing buttons and watching status bars…Then after pressing the buttons and watching the status bars hundreds or even thousands of times I started to wonder, “Is this all there is to IT, pressing buttons and watching status bars?” I then started to explore the ways I could make my job easier and sort of offload some of the repetitive tasks. Throughout school, I’ve had a fair amount of math and programming. I understood algorithms and loops. So, like any good IT professional of the time, I started writing VBScripts. Oh yeah, like He-Man use to say, holding up his sword “I have the power!” VBScript was awesome. I could reference libraries for reuse just like writing Visual Basic programs, but in a script with no compile and packaging.

A few years have passed and thousands of lines of VBScript under my belt. Since then VBScript has gotten a bad rap and for good reason. It served its purpose and now a new kid is on the block. Powershell.

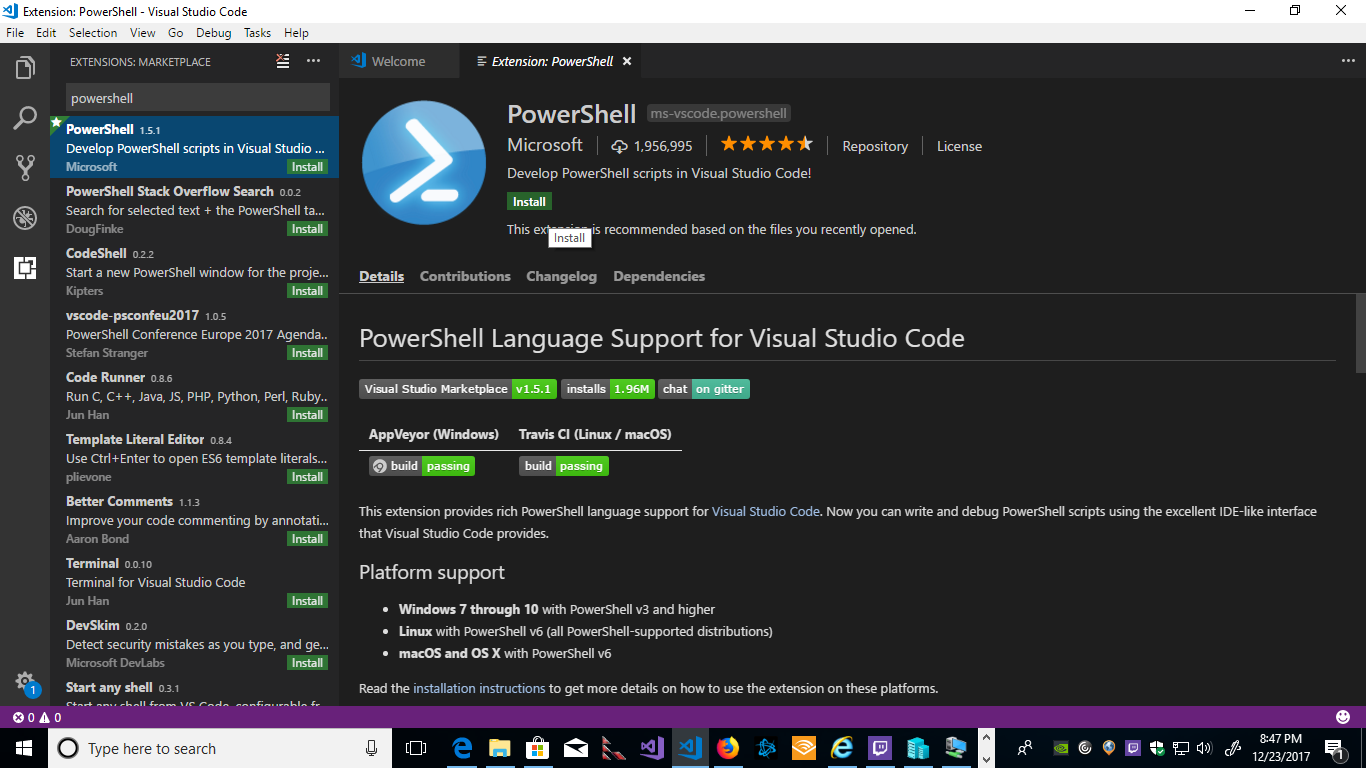
So, before I decided to write this book I sat down and started to make a list of all the things I do with Powershell or how Powershell makes my life easier. After all, if Powershell isn’t helping to satisfy your internal “lazy bastard,” (as my friend Harry calls his need to automate things) what’s it doing for you anyway? Powershell can be used to remotely manage endpoint (pc and non-pc). It can be used for file I/O. It can be used to Rickroll your friends…oh wait…I would never do that. However, there are plenty of books to teach you the basics of Powershell. This book doesn’t do that. This book is for individual who have a basic understand of Powershell already. You understand how to declare variables, write loop logic, and run your scripts. Powershell unpractical? Of course not. But this book isn’t for the practicality of Powershell. This book is to express the elegance and flexibility behind Powershell, to show the things us “lazy bastards” may be missing, or what may be considered “unpractical” because, quite frankly, you didn’t know Powershell could do that.

Things covered in this book may not be particular to your job function but the hope is that it helps and gives you a greater appreciation for the language of Powershell that I appreciate so much.

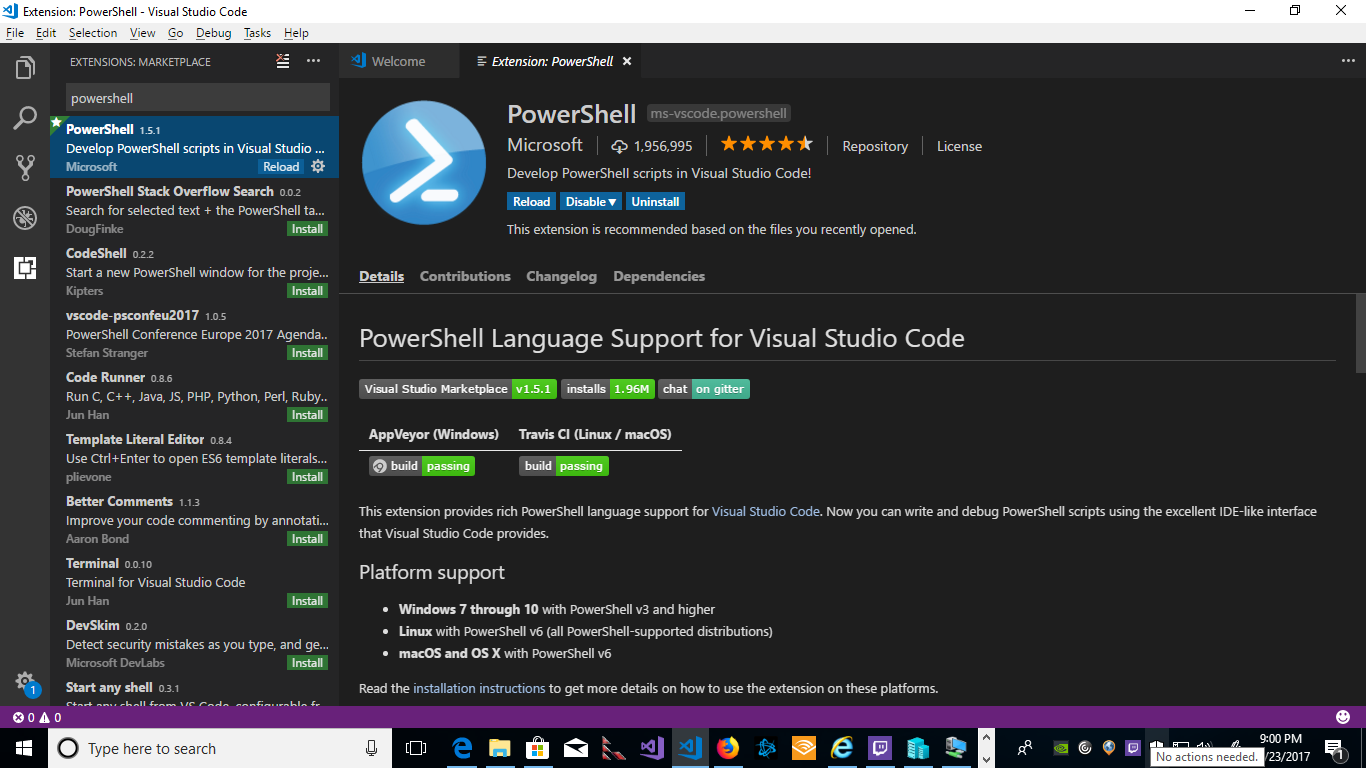
## Setting Up Your Environment

Just like any good artist you’ll need a pallet to paint on. Before you start writing your Powershell you need to make sure you have an Integrated Development Environment (IDE) you are comfortable with. I’m not a Microsoft employee and have no biases. Its simply up to you. However, throughout this book we will be using Microsoft Visual Studio Code on Windows 10. Visual Studio Code will run on Windows, Mac, and Linux. You can download Visual Studio Code from the link <https://code.visualstudio.com/>

Once you have Visual Studio Code installed you should install the Powershell plugin. You can do this by launching Visual Studio Code, clicking on the Extensions icon and searching for “powershell.” Make sure you are installing the plugin authored by Microsoft and then click Install.



Once the Powershell plugin has been installed you will see a “Reload” button. Go ahead and click reload.



This will recycle your Visual Studio Code IDE with the Powershell extension enabled. Having the Powershell extension enables us to use the F5 key to run and debug as well as use the Powershell command prompt from within our Visual Studio Code session.

## Powershell Modules

So, lets first talk about Modules. Powershell Modules are the jelly of the month club, the gift that keeps on giving. Modules are very important. I’m going to take a bit of time in this Chapter to really dig into modules because we will use this knowledge of modules as the book goes on. I promise if you take the time to really understand Powershell modules it will pay off.

Powershell Modules are Powershell code that is written and then packaged in a format that allows you to reuse your code and/or share the code with someone else. Modules allow you to share your work with the world (or at least your colleagues) so that others don’t have to reinvent a solution to a problem you’ve already solved. Then once you are completely satisfied with your working module you can publish it to Github, but more on that later.

If you don’t “get” Modules the first time around please reread the chapter until you are able to do the following:

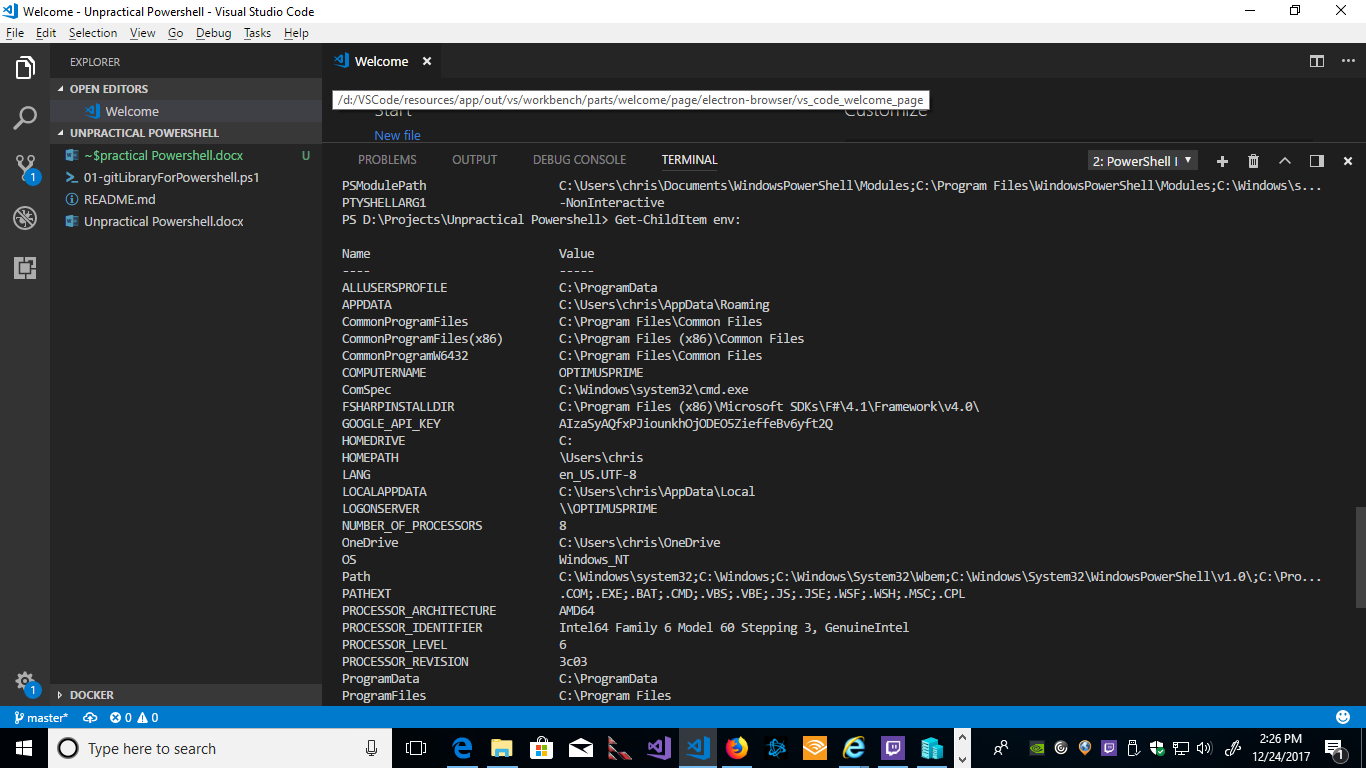
1. Know where in the directory structure your modules should go
2. Know what kind of files make up a module
3. How to load and unload a module
4. How to call a module

Modules can have a few different parts and can be very complex or really simple. Modules can be comprised of the following but not limited to:

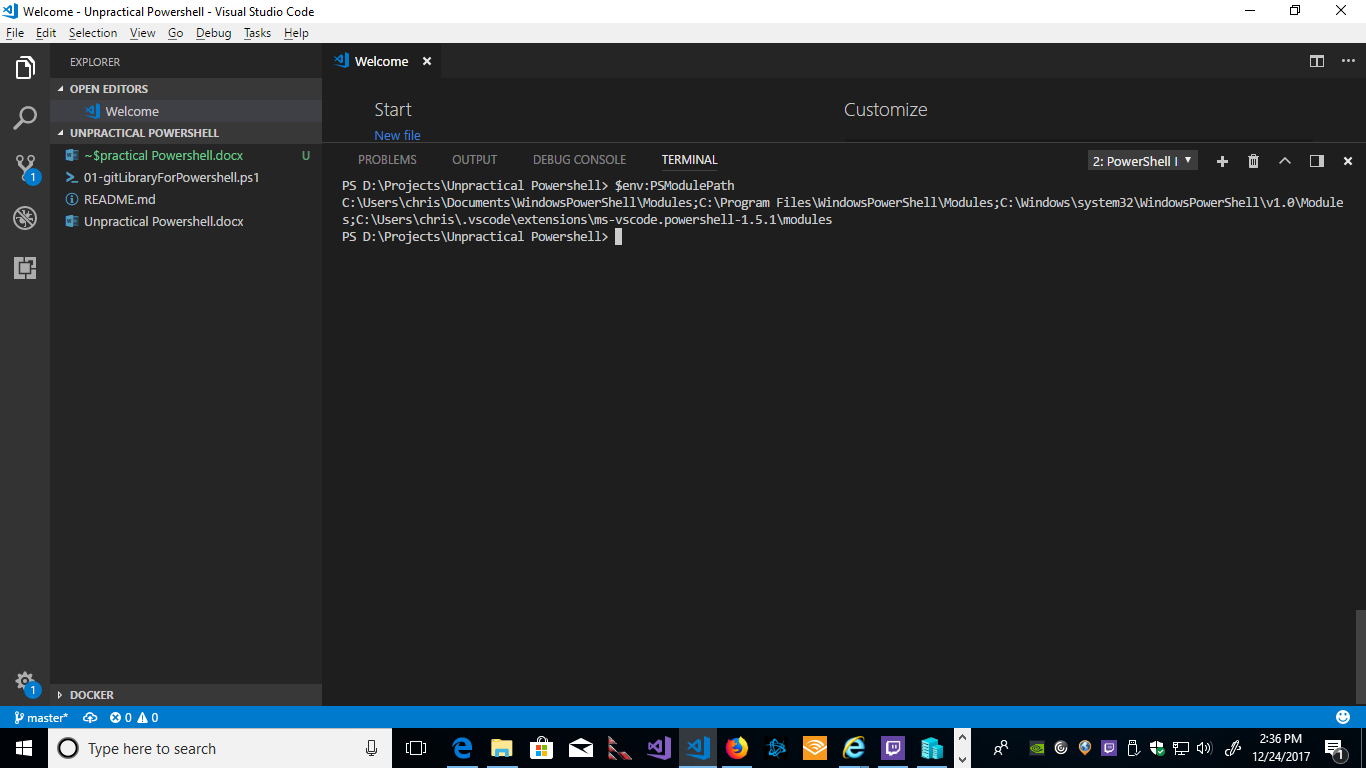
1. A directory that contains all of the above content, and is located where PowerShell can reasonably find it.
2. Some sort of code file – usually either a PowerShell script or a managed cmdlet assembly.
3. Anything else that the above code file may need, such as additional assemblies, help files, or scripts.
4. A manifest file that describes the above files, as well as stores metadata such as author and versioning information.

(Microsoft)

Let’s first explore how Microsoft stores the Modules you may access from your Powershell. Within a Powershell session there are a certain number of environmental variables Powershell uses to run your code. If you would like to see them you can run the following in your Powershell session Get-ChildItem env:



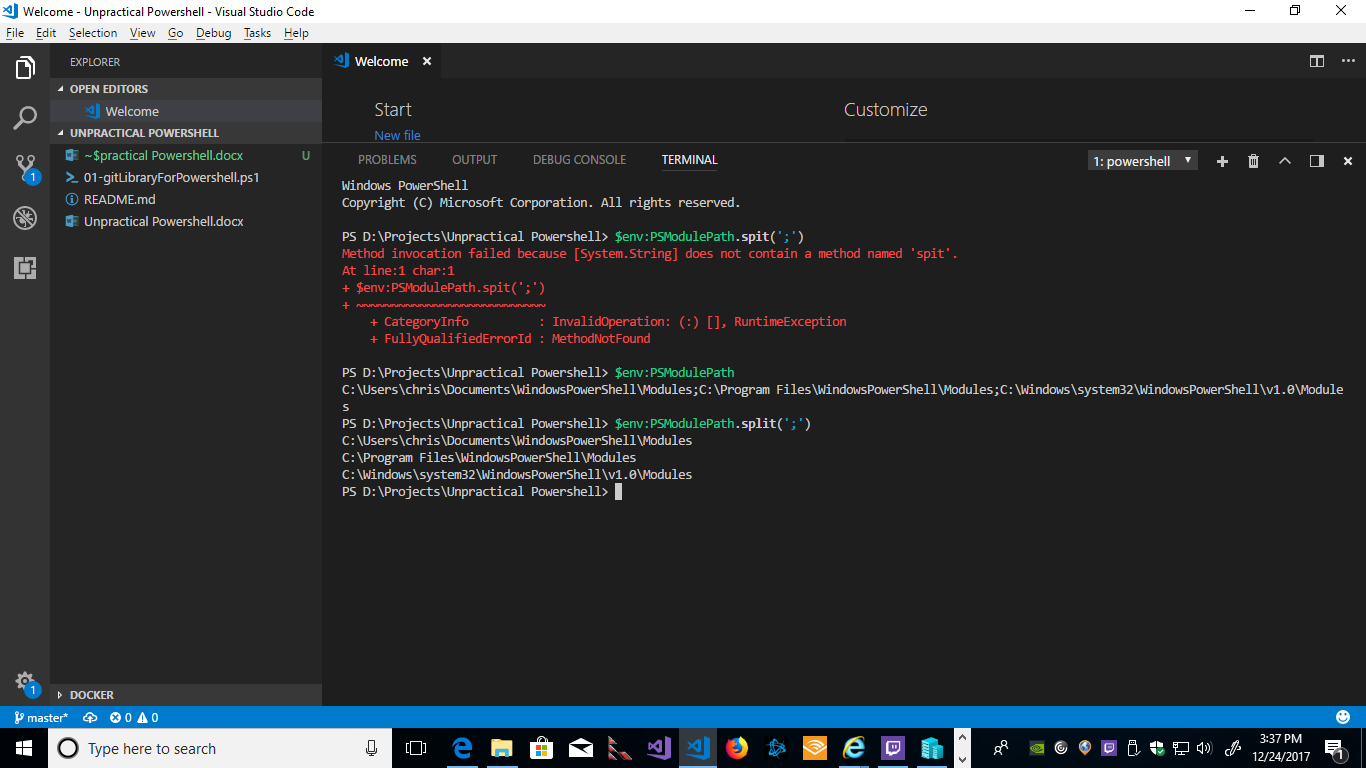
This is a truncated list of the variables on my computer. There are quite a bit more on this list but the one in particular we are concerned with is the PSModulePath variable. You can view by running the following $env:PSModulePath



Notice this time to access the individual variable I put the $ in front of env to access the PSModulePath variable.

This is kind of hard to read though. Let’s use the split method to separate the different directory structures out so they are easier to read. Run the following instead.

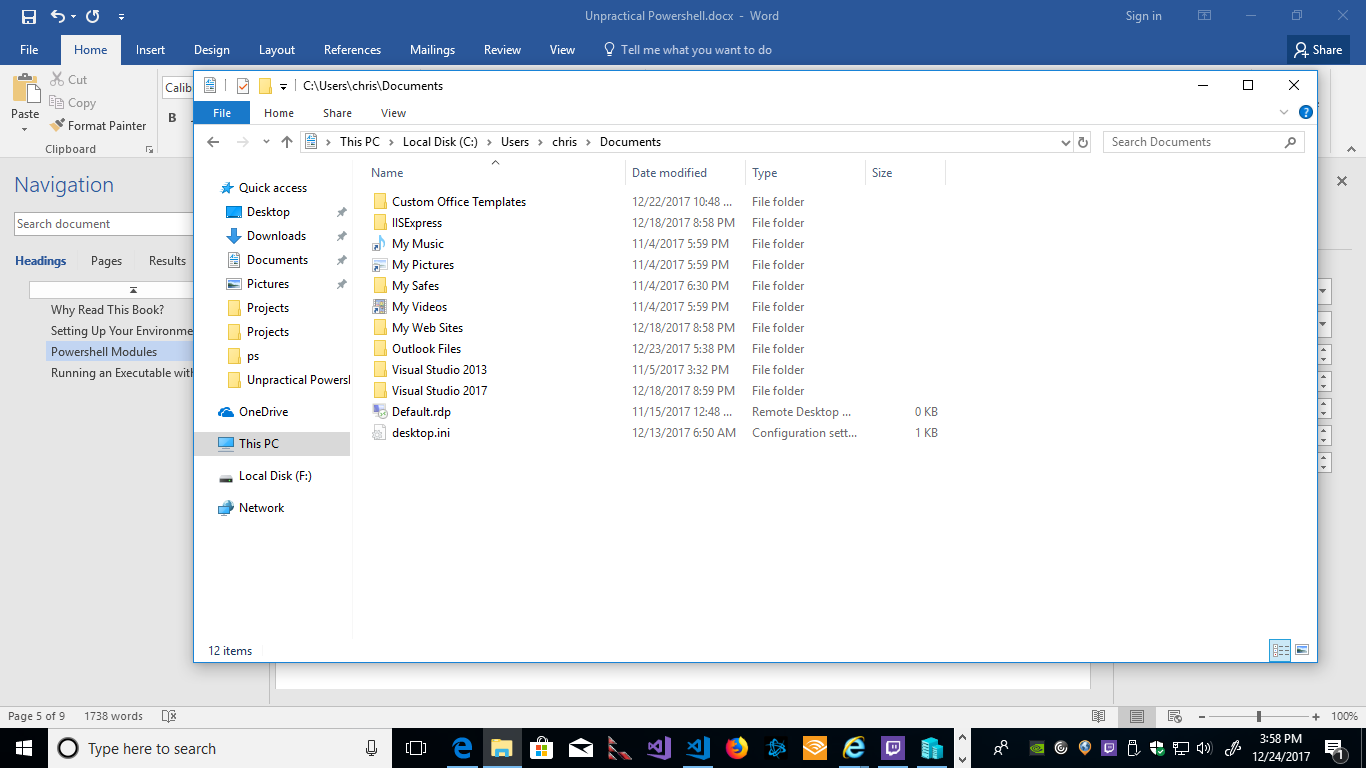
$env:PSModulePath.split(‘;’)



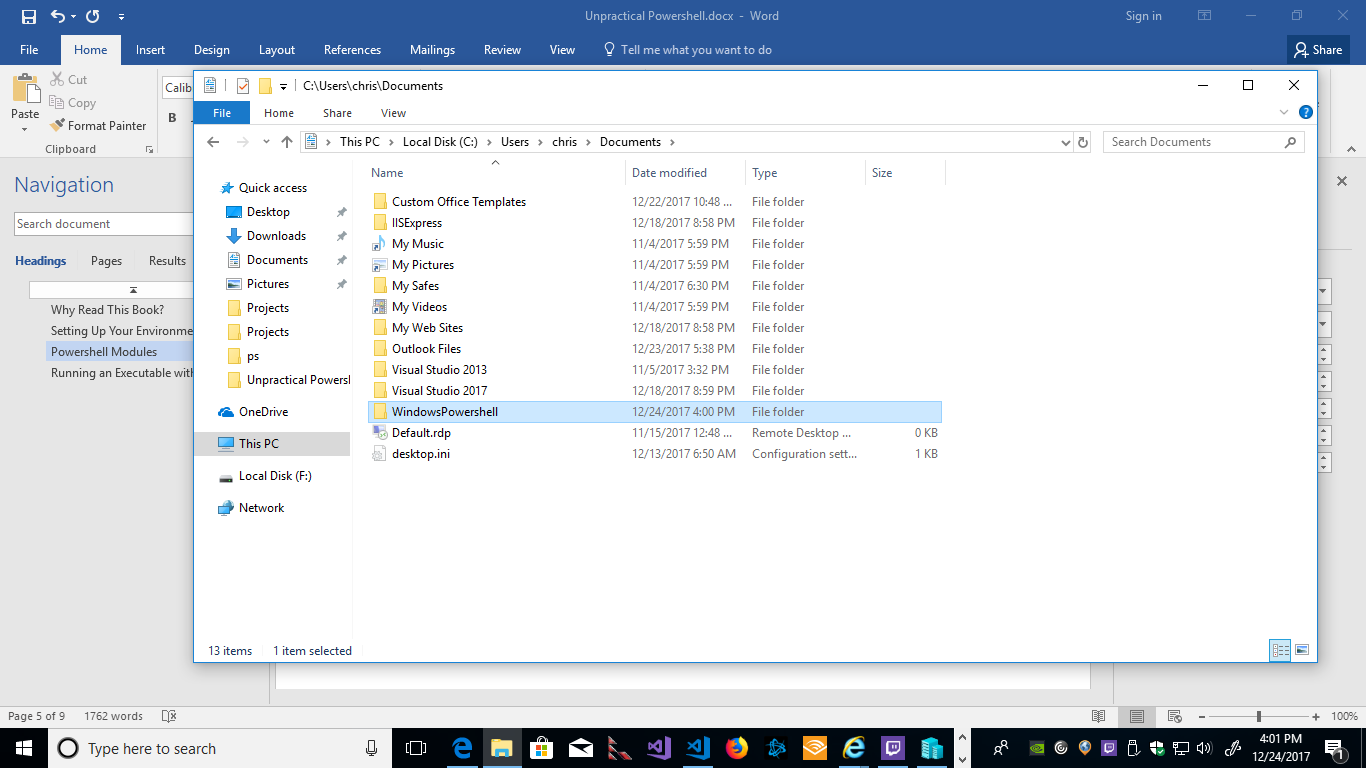
*Note: the split() method is a method of a string object in Powershell. It will read the string and replace the character specified with a new line character. It helps with readability in certain cases.*

There, that’s better. This is the list of directories Powershell will iterate through to find a function reference not part of the Powershell run time. Let’s take a look at the contents of the first directory in the list C:\Users\Chris\Documents\WindowsPowershell\Modules. This is my personal profile directory on my local Windows 10 machine. The first line in the list on your machine will look something similar to the below.

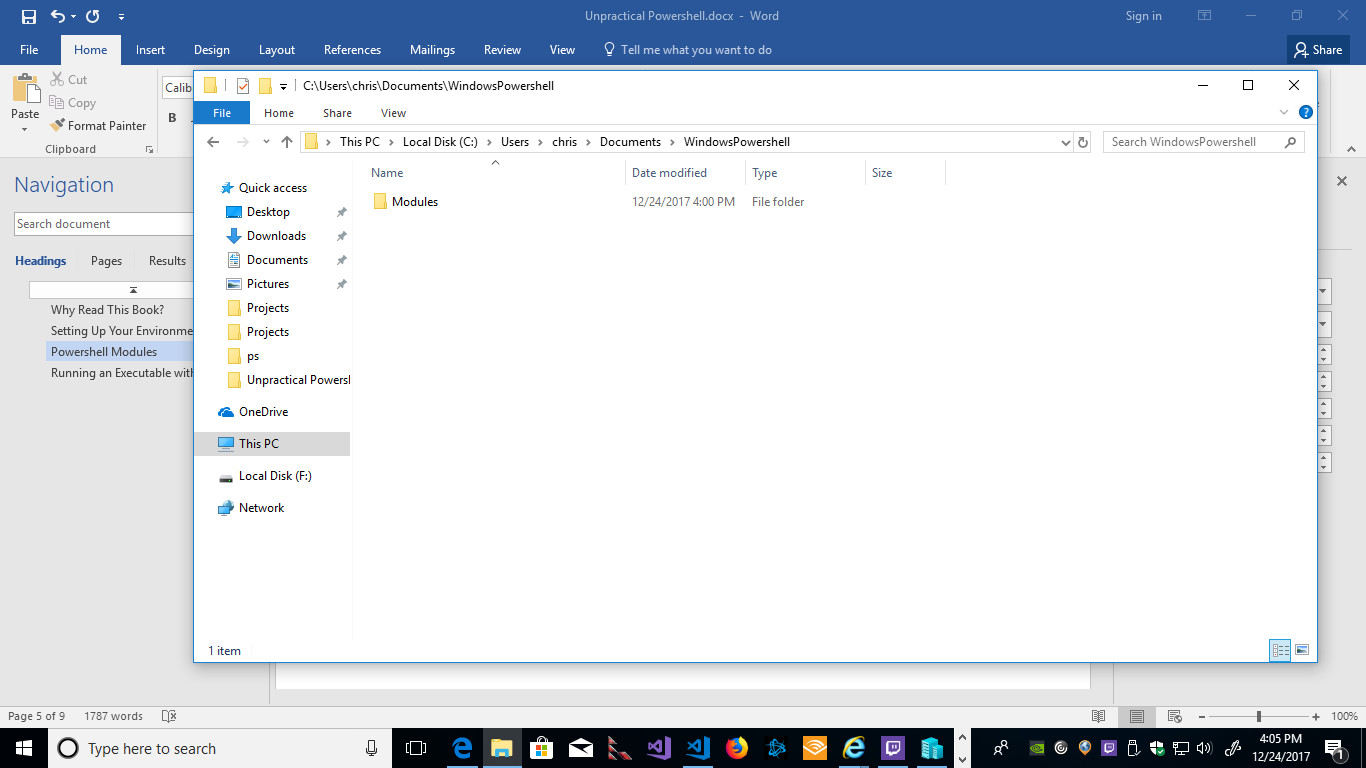
C:\users\<your username>\Documents\WindowsPowershell\Modules



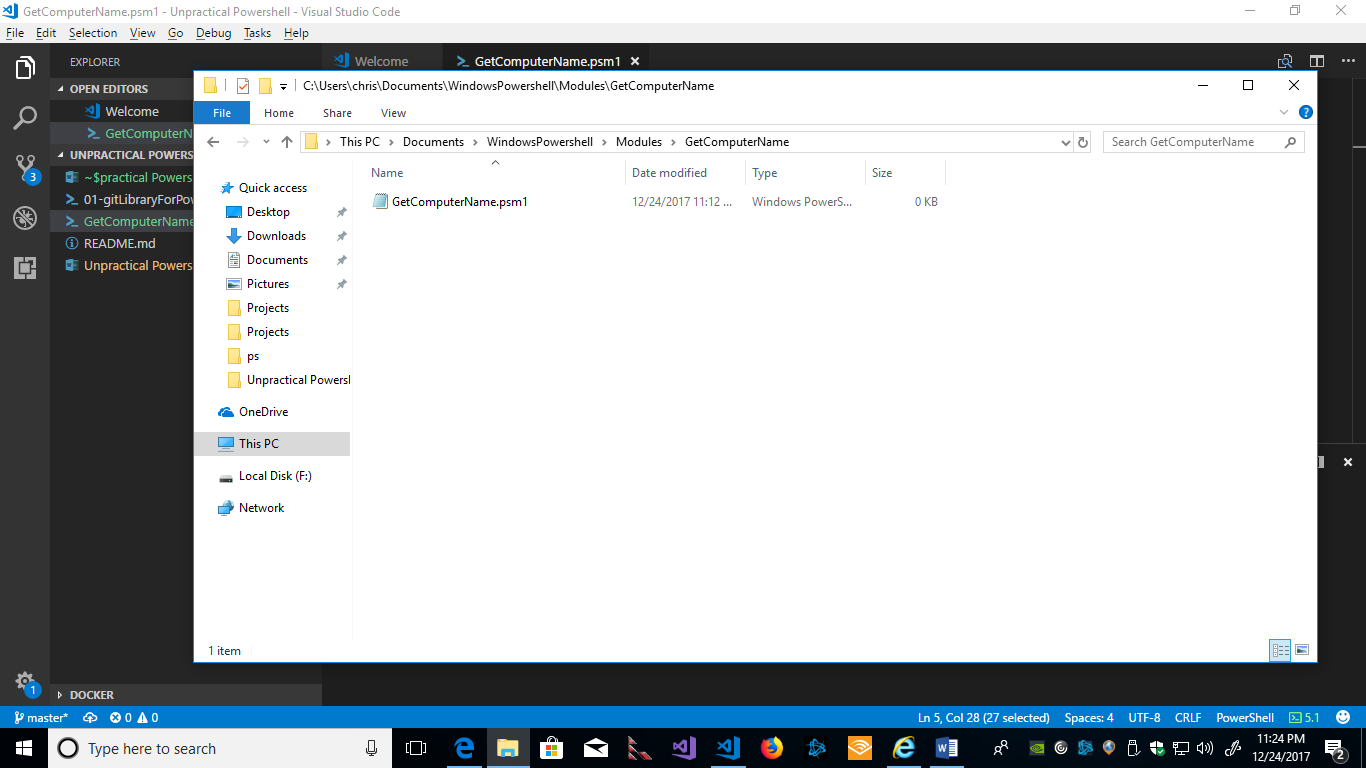
Wait a minute! The directory structure isn’t there? What gives? It turns out that by default this directory structure doesn’t exist so we need to create it. Lets create the **WindowsPowershell** directory under our profile **Documents** folder.



Next let’s create a directory called **Modules** in the **WindowsPowershell** directory we just created. This **Modules** directory will contain the custom modules we will write. By the way, its best practice to always put your custom modules here under your Windows profile directory.



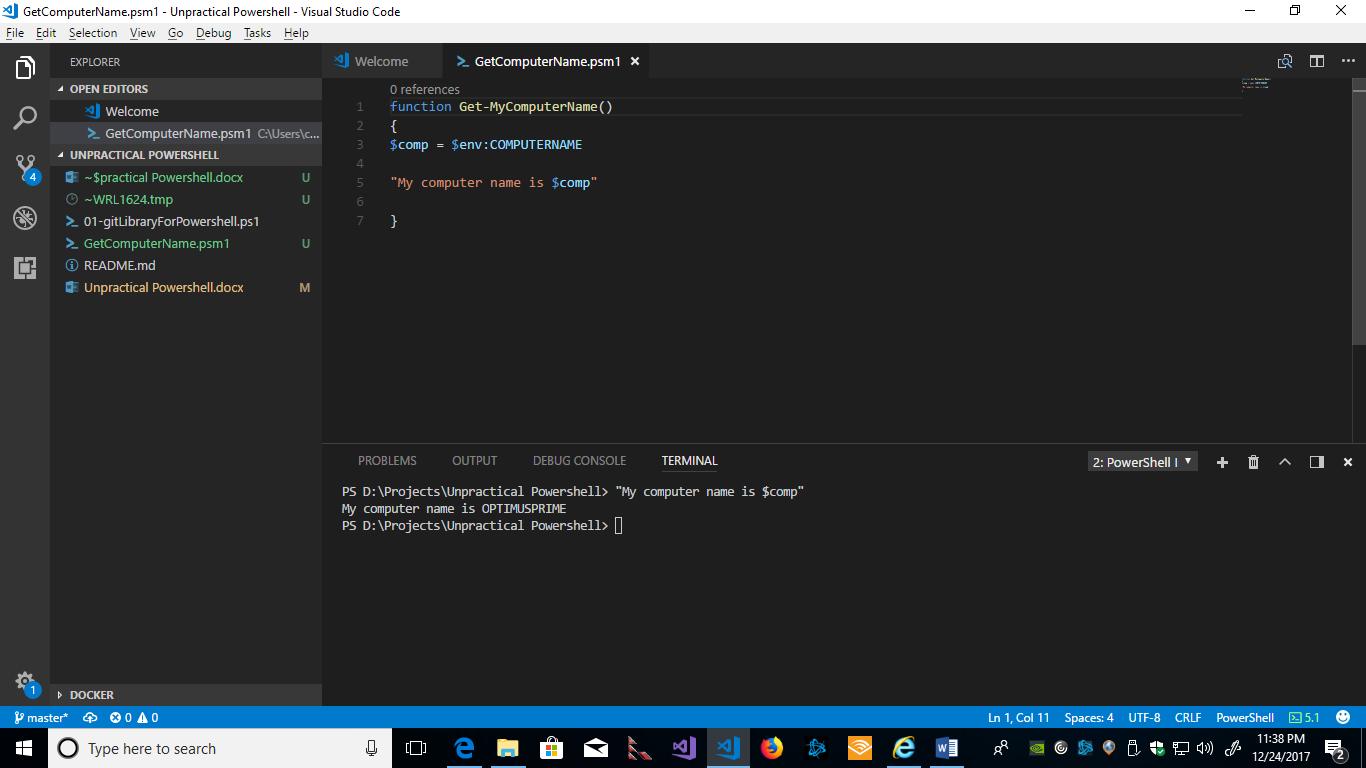
Now that we have our custom Modules directory structure in place lets talk about what makes up a Powershell Module. A very basic Powershell Module will be comprised of a PSM1 file within a directory of the same name as the PSM1 file you put your custom code in. For example, the structure for a module may look like the below (and yes, I’m writing this book on Christmas Eve).



Let’s create this example together. Create a directory named GetComputerName in your Modules directory. Then create a text file in the directory call GetComputerName.psm1.

*Note: Make sure the file extension is really PSM1. If you aren’t showing file extensions by default now is a good time to change this in your Folder Options.*

Let’s modify the file by adding the following code to the file and save it.



Now we have our .PSM1 file saved in our **GetComputerName** folder in our **Modules** directory. We can now load our new Module and run the **Get-MyComputerName** function.

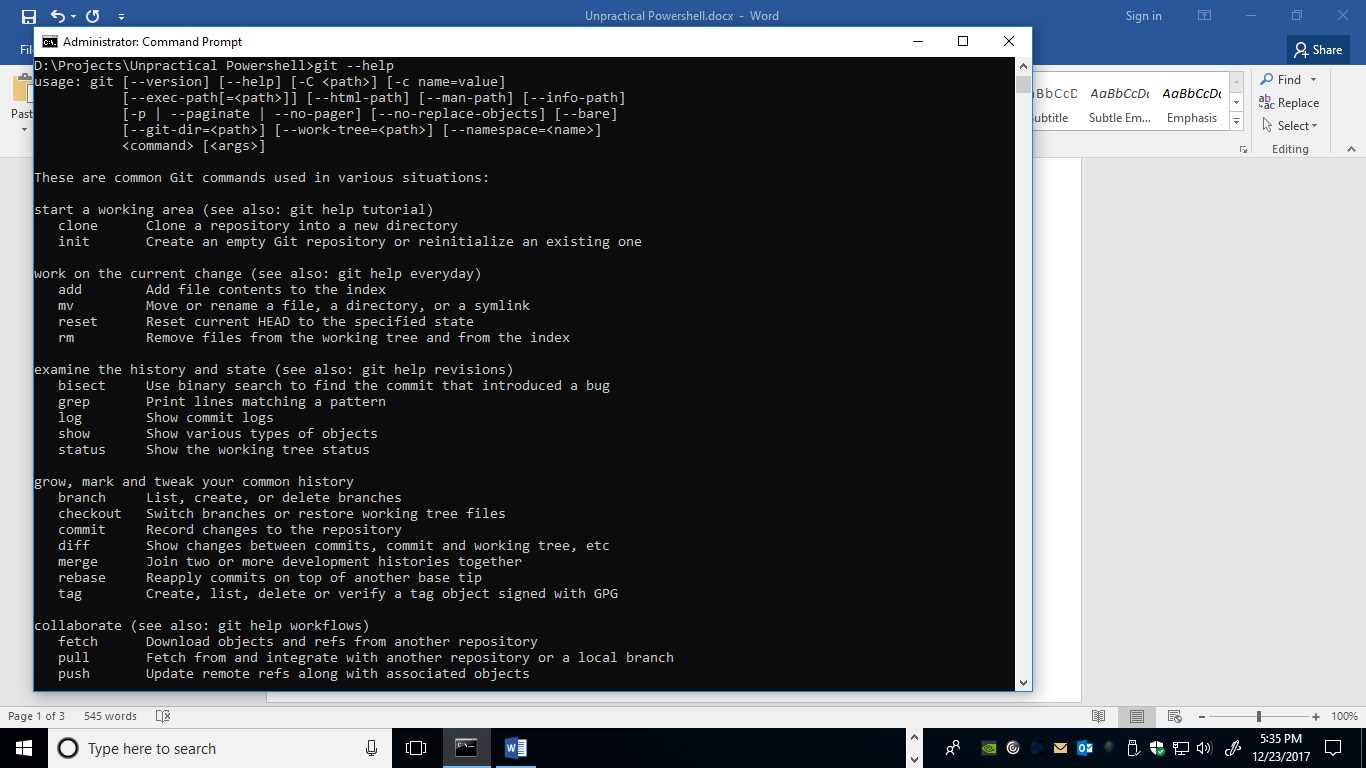
We do this by running the **Import-Module** command and then specifying which module we want to load into our session, in this case the **GetComputeName** module.

## Running an Executable with Parameters Using Powershell

One of the common things you may find yourself doing is calling an executable from a command line. However, there may be executables you might want to whittle down command line arguments for that you need to run often.

Our first example is actually going to be dealing with Git, particularly the Git executable. I won’t go into detail what Git is, but that it is a source control for the software you write. If you write any software you should be using it, whether in a team or by yourself. Using Git helps develop your skillset as a professional IT person. If that doesn’t appeal to you that’s fine. You’ll still get the jest of what we are trying to do from the example below.

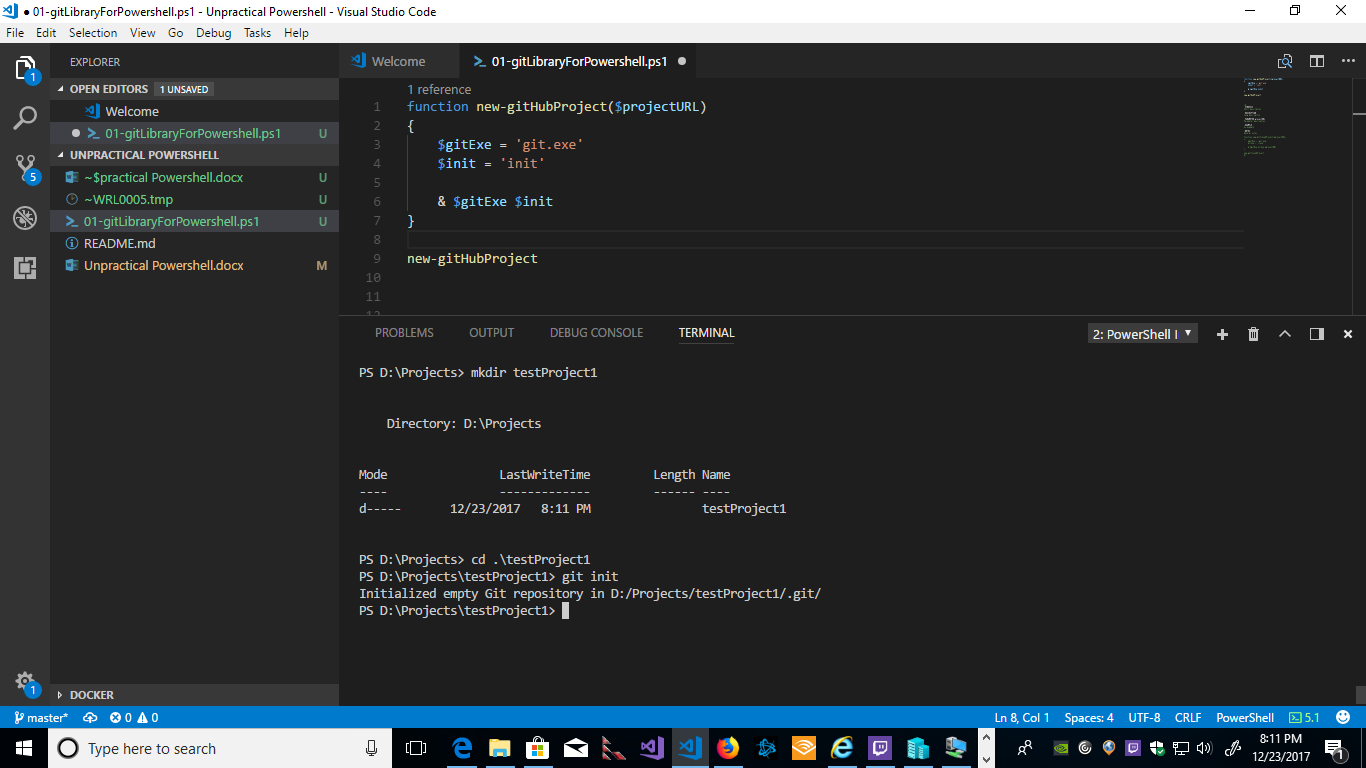
First, if you don’t have Git you can download the version for your operating system at github.com. You need to make sure that when you run git --help from a command line you get something similar to the below.



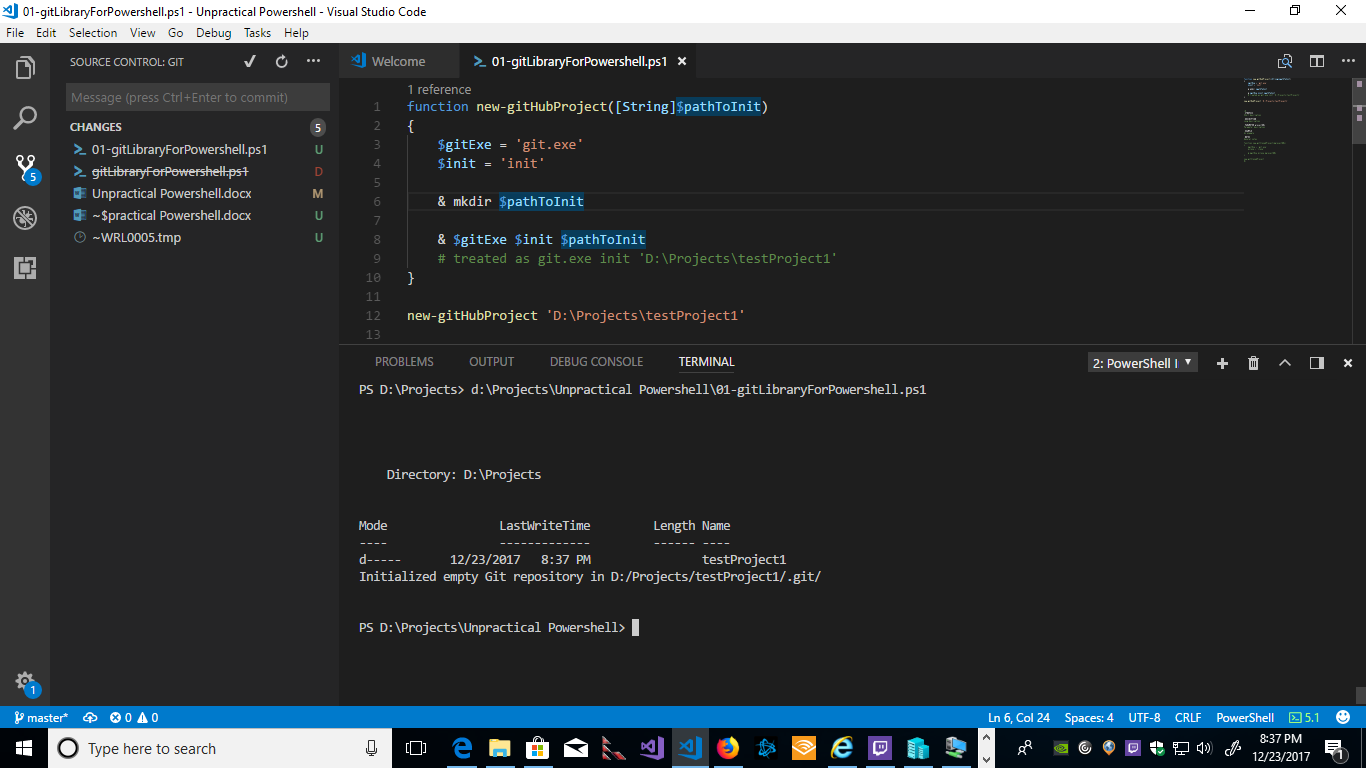
This ensures that Git is running from a command line prompt. You should also make sure the Git.exe is defined in your system path for this example. This ensures that running Git from any folder will provide consistent results.

Running an executable from Powershell can be a bit confusing depending on the format of the parameters being passed and the spaces that may be used with the executable parameters. One way to combat this is by prefixing your code with the ampersand sign ‘&’ or what is considered the *call operator* (Microsoft) in Powershell. The call operator treats the entire string as a single command.

For this example, I need to create a directory to initialize as a git project on my local machine. To create my test project directory from the command line I’m going to run the command mkdir testProject1. This will create the new directory in my current Projects directory. Then I would cd testProject1 I would then run git init. The syntax would look like the following:

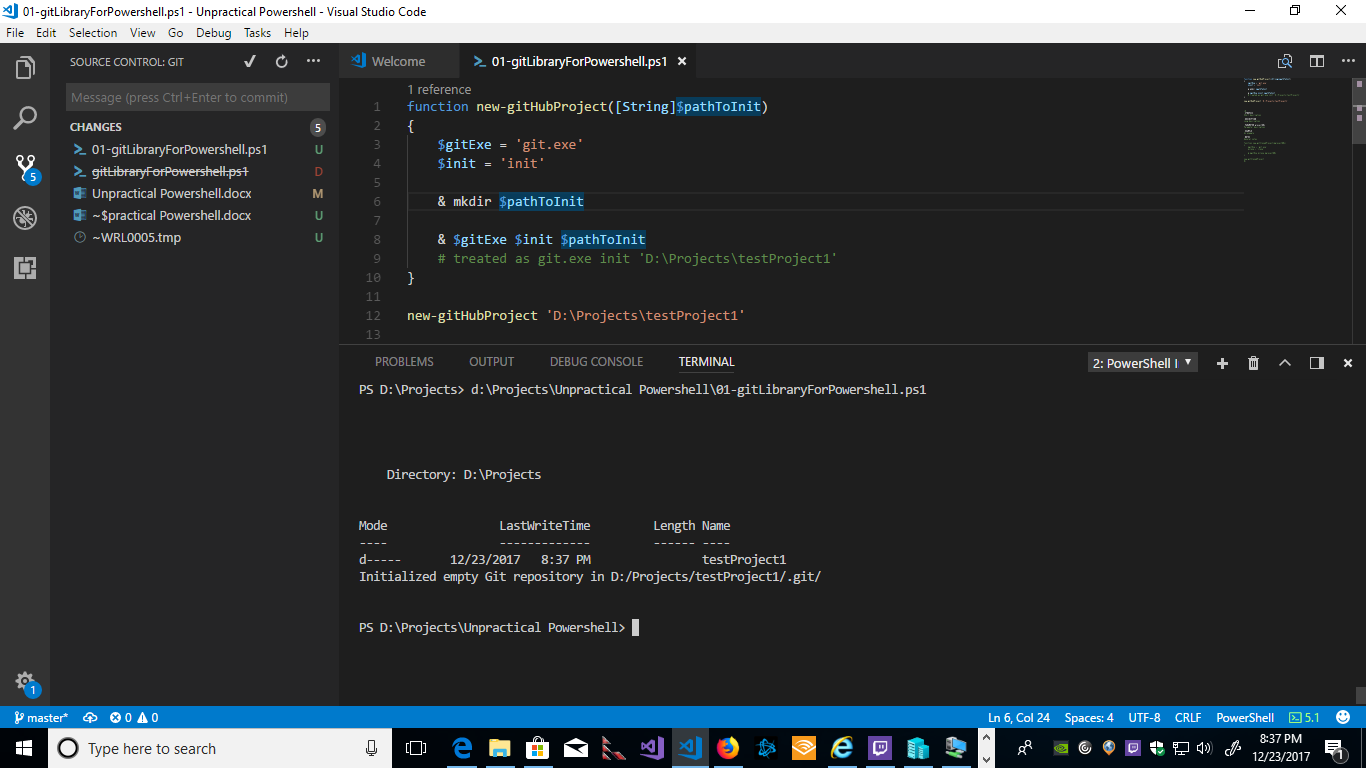


However, in this case I’m going to create an Powershell function to do this for me. The syntax looks like this



So, what’s going on here? On line 1 I’m creating the function call and the parameter of the path I want to initialize as my new Git project. On line 3 I’m declaring the string of the executable name I want to run. On line 4 I’m defining the string I want to pass the git.exe as my parameter. On line 6 I’m calling the command mkdir the same way I’m going to call the git.exe to create my new directory.

Finally, on line 9 I’m calling the function. So that the result looks like the following:



This is typically done by using the command git init from a command line within the directory you want to initialize as the git project.

So, for example if we want to clone an existing project from Github.com we can use the following syntax.

Functions and Parameters

Try/Catch/Trap Error Handling

Running commands with parameters from Powershell

Backup from Powershell

Accessing WMI/CIM objects via Powershell

Containers from Powershell

Generating User Interfaces with Powershell

Working with the Web

Invoke-WebRequest

function test-url ($url) {

$HTTP\_Request = [System.Net.WebRequest]::Create($url)

# We then get a response from the site.

$HTTP\_Response = $HTTP\_Request.GetResponse()

# We then get the HTTP code as an integer.

$HTTP\_Status = [int]$HTTP\_Response.StatusCode

If ($HTTP\_Status -eq 200) {

Write-Host "Site is OK!"

return $true

}

Else {

Write-Host "The Site may be down, please check!"

return $false

}

# Finally, we clean up the http request by closing it.

$HTTP\_Response.Close()

}