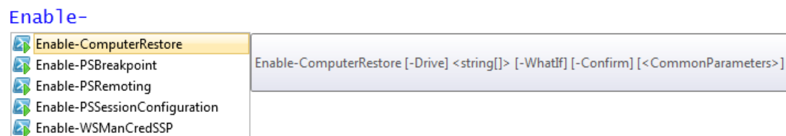
Last updated: November 29, 2011

# Windows PowerShell ISE

Windows PowerShell Integrated Scripting Environment (ISE) was first introduced in Win7. This short document describes the new features which have been added to the ISE since then, along with Out-GridView enhancements and the brand new Show-Command feature.

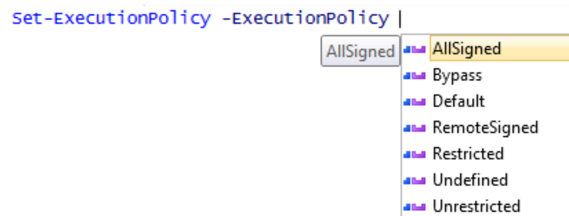
## Intellisense

ISE Intellisense is similar to Visual Studio's Intellisense. While tab completion allows you to cycle through the options, Intellisense allows you to select from a drop-down list of options.




Intellisense automatically displays a list of options when you enter:

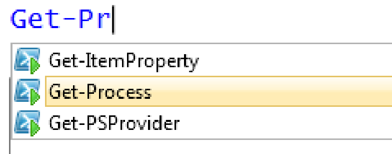
- "-" (dash) after a verb, as in `Get-` or before a parameter name, as in `Get-Process -`
- "." (period) after an object, as in `$host.`
- "::" (double colon) after a type, as in `[int]::`
- "\" (backslash) for providers, as in `C:\`
- " " (space) after parameters (enumerations & ValidateSet only), as in:




Notes:

- Expect some delay the very first time you use Intellisense in the ISE, so you should probably wait after you type the dash "-"
- If you dismiss Intellisense (using "Esc" for example), you can engage Intellisense again by pressing Ctrl+Space, or by selecting "Start Intellisense" from the "Edit" menu.

Intellisense uses "smart filtering" to reduce the options in the drop-down list, based on what you are typing, as in the following example:




While Get-Process is highlighted by default (because it starts with “Pr”), Get-ItemProperty and Get-PSProvider are also included in the list. This makes noun searching much easier.

To select an option from the drop-down list, you can use the mouse. Alternatively, you can use the keyboard’s arrow keys, plus:

- “Enter”
- “Tab”

If you don’t want the “Enter” key to select Intellisense, you can set the following options in the object model to `$false`, or use the Options dialog.

```
# Set this if you don’t want <Enter> to select Intellisense in the Console Pane
$psise.Options.UseEnterToSelectInCommandPaneIntellisense = $false

# Set this if you don’t want <Enter> to select Intellisense in the Script Pane
$psise.Options.UseEnterToSelectInScriptPaneIntellisense = $false
```

Intellisense was designed to allow a user to use tab completion seamlessly, as if Intellisense didn’t exist at all. However, you may also explicitly turn off Intellisense altogether in the Console Pane, Scripting Pane, or both, using the following configuration options:

```
# Turn off Intellisense in the Console Pane
$psise.Options.ShowIntellisenseInCommandPane = $false

# Turn off Intellisense in the Script Pane
$psise.Options.ShowIntellisenseInScriptPane = $false
```

# Snippets

Snippets allow insertion of text right from the editor (or console pane), by pressing Ctrl+J or selecting “Start Snippets” from the “Edit” menu. There are three types of snippets:

- Default snippets that ship with the ISE
- User-defined snippets
- Module-based snippets

## Default Snippets

The ISE ships with a set of default snippets, to help the beginner, and as a convenience for the advanced user. These snippets can be disabled (if you prefer to create your own, for example), using:

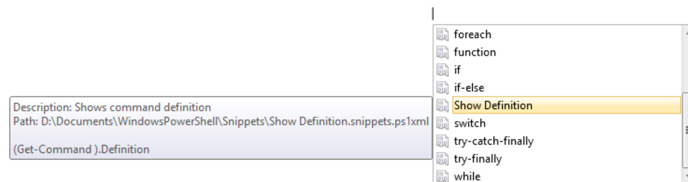
```
$psise.Options.ShowDefaultSnippets = $false
```

## User-Defined Snippets

You can create your own snippets and add them to the snippets list using the `New-Snippet` cmdlet:

```
New-Snippet -Title "Show Definition" -Description "Shows command definition"
             -Text "(Get-Command ).Definition" -CaretOffset 13
```

This example will add a “Show Definition” snippet to the snippets list, and will insert the following text when selected: `(Get-Command ).Definition`




Notes:

- The caret is placed 13 characters after the start of the snippet text (ready for the user to enter the command name). This is what the –CaretOffset parameter does
- Example usage: `(Get-Command prompt).Definition`
- To see all user-defined snippets, use: `Get-Snippet`
- To remove a snippet, go to the folder which contains all user-defined snippets (you can get this from `Get-Snippet`) and delete the associated snippet file.

If you prefer to hand-craft your snippets instead, just create an XML file like the one in the following example, and copy it to the snippets “home” folder, which you can get using one of the following methods:

- `Join-Path (Split-Path $profile.CurrentUserCurrentHost) "Snippets"`
- Use `Get-Snippet` and copy the displayed path ☺

Any snippets in the snippets home folder will be loaded automatically by the ISE when it starts.  If you don't want to place your snippets in the home folder, you can place them anywhere (C:\temp for example), but then you must load them explicitly in our ISE profile using:

```
$psise.CurrentPowerShellTab.Snippets.Load("C:\Temp", $true)
```

Snippets files have an extension of .snippets.ps1xml and have the following structure:

```
<?xml version='1.0' encoding='utf-8' ?>
<Snippets  xmlns='http://schemas.microsoft.com/PowerShell/Snippets'>
<Snippet Version='1.0.0'>
<Header>
<Title>TryCatchFinally</Title>
<Shortcut></Shortcut>
<Description>Exception handling using try, catch, finally</Description>
<Author>Microsoft Corporation</Author>
<SnippetTypes>
<SnippetType>Expansion</SnippetType>
</SnippetTypes>
</Header>

<Code>
<Script Language='PowerShell' CaretOffset='0'>
<![CDATA[try
{
}
catch
{
}
finally
{
}]]>
</Script>
</Code>

</Snippet>
</Snippets>
```

Where:

- Title: The name which appears in the snippets drop-down list
- Description: The description which appears in the snippet tooltip
- Cdata: The tag which contains the actual snippet text

## Module-based Snippets

To load all module-based snippets (which only works for modules which have already been imported using Import-Module), use:

```
$psise.CurrentPowerShellTab.Snippets.LoadFromImportedModules()
```

If you're shipping a module with snippets, create a "Snippets" folder under the module folder, and place the snippet files into that folder.

# Editor Enhancements

## Brace Matching & Highlighting

When you place the caret right before an opening brace (parentheses, square brackets, or curly braces), or right after a closing brace, the editor highlights both braces, and you can "jump" the caret from one brace to the other by pressing Ctrl+] or using the "Go to match" menu item in the "Edit" menu.

```
122  function clearOutput
123  {
124      $psISE.CurrentPowerShellTab.Output.Clear()
125  }
126  $psise.CurrentPowerShellTab.AddOnsMenu.Submenus.Add("Clear Output",{clearOutput},"F12")
```

## Outlining

This feature allows collapsing or expanding a region in the code, indicated by outlining lines in the left margin as shown:

```
190  function insertIf
191  {
192      $l = $psise.CurrentFile.Editor.CaretLine
193      $c = $psise.CurrentFile.Editor.CaretColumn
194      $x = ''
195      if ($c -ne 0)
196      {
197          $x = ' ' * ($c - 1)
198      }
199      $psise.CurrentFile.Editor.InsertText("if()`n" + $x + "{`n" + $x + "}")
200      $psise.CurrentFile.Editor.SetCaretPosition($l, $c + 3)
201  }
202  $menuItem = $psISE.CurrentPowerShellTab.AddonsMenu.Submenus.Add("Insert if", {InsertIf}, 'Ctrl+S
```

When you hover over the "minus" sign, the region to be collapsed will be highlighted in a darker color:

```
190  function insertIf
191  {
192      $l = $psise.CurrentFile.Editor.CaretLine
193      $c = $psise.CurrentFile.Editor.CaretColumn
194      $x = ''
195      if ($c -ne 0)
196      {
197          $x = ' ' * ($c - 1)
198      }
199      $psise.CurrentFile.Editor.InsertText("if()`n" + $x + "{`n" + $x + "}")
200      $psise.CurrentFile.Editor.SetCaretPosition($l, $c + 3)
201  }
202  $menuItem = $psISE.CurrentPowerShellTab.AddonsMenu.Submenus.Add("Insert if", {Inser
```

When you click the "minus" sign, the region collapses as shown:

```
190  function insertIf
191  {...}
202  $menuItem = $psISE.CurrentPowerShellTab.AddonsMenu.Submenus.Add("Insert if", {InsertIf}, 'Ctrl+S
```

Braces provide automatic outlining. In addition, you can use #region and #endregion to create an arbitrary outlining region:

```
2  #region stuff
3  Get-Date
4  Get-Process
5  Format C:
6  #endregion
```

When collapsed:

```
2 ⊞#region stuff...
7
```

You can turn this feature off, using the View menu's "Show Outlining (Regions)" option, or using the object model: `$psise.Options.ShowOutlining = $false`

Outlining is especially useful for XML files.

## Error Indication

Parse errors are indicated using squiggle marks. When you hover over the squiggle, a tooltip displays the warning message

```
3  if
   Missing '(' after 'if' in if statement.
```

Error indication works for XML files as well.

## Zoom

Two parameters affect the perceived size of the font on the screen: the actual font size and the zoom level

- You can set the font size using `$psise.options.FontSize`, or through the Options dialog. This font size is used when you copy-and-paste to other applications.
- The zoom level can be set using the zoom slider (in the bottom right corner of the ISE window), using Ctrl++ (press Ctrl and plus at the same time), through: `$psise.options.Zoom`, or through the Options dialog

## Rich Copy

When you copy text to the clipboard in the ISE, the font size and color, as well as the background color information is copied to the clipboard, allowing you to paste rich text into applications which can handle it, such as Word or Outlook.

## Block Select

A user can select a block of text using the Alt key and the mouse (or Alt + Shift + Arrow keys):

```
176  # Black theme
177  $psise.Options.OutputPaneBackgroundColor = "Black"
178  $psise.Options.OutputPaneTextBackgroundColor = "Black"
179  $psise.Options.OutputPaneForegroundColor = "White"
180  $psise.Options.VerboseForegroundColor = [System.Windows.Media.Color]::FromRgb(199,199,99)
```

Once a block has been selected, you can copy, cut, delete, or "fill" this block. For example, typing '$xyz' (or pasting '$xyz' from the clipboard) will have the following "fill" effect:

```
176  # Black theme
177  $xyz.Options.OutputPaneBackgroundColor = "Black"
178  $xyz.Options.OutputPaneTextBackgroundColor = "Black"
179  $xyz.Options.OutputPaneForegroundColor = "White"
180  $xyz.Options.VerboseForegroundColor = [System.Windows.Media.Color]::FromRgb(199,199
```

Notice the thin blue vertical line? This is a zero-width block selection, which enables the “fill” effect. In other words, you can append / prepend block text, instead of just replacing it. In the next example, manually create a zero-width block selection, just to the left of the text:

```
176  # Black theme
177  $psise.Options.OutputPaneBackgroundColor = "Black"
178  $psise.Options.OutputPaneTextBackgroundColor = "Black"
179  $psise.Options.OutputPaneForegroundColor = "White"
180  $psise.Options.VerboseForegroundColor = [System.Windows.Media.Color]::FromRgb(199,1
```

Now, enter a single hash character (#):

```
176  # Black theme
177  #$psise.Options.OutputPaneBackgroundColor = "Black"
178  #$psise.Options.OutputPaneTextBackgroundColor = "Black"
179  #$psise.Options.OutputPaneForegroundColor = "White"
180  #$psise.Options.VerboseForegroundColor = [System.Windows.Media.Color]::FromRgb(199,
```

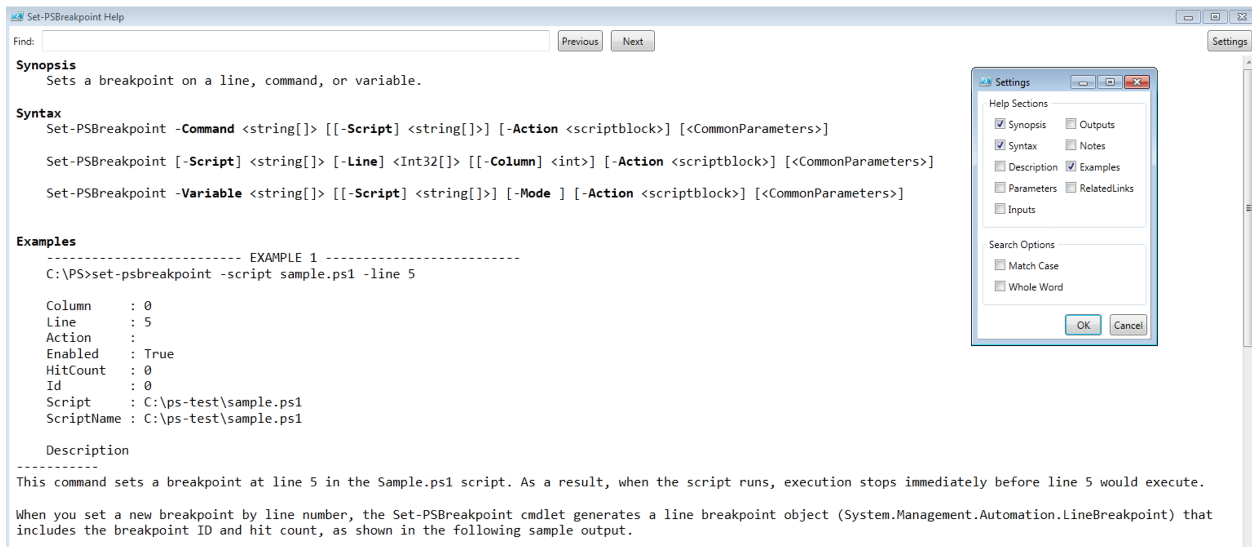
Other Editor Enhancements

- XML syntax coloring
- Hiding the editor’s line numbers using the “Hide Line Numbers” option from the “View” menu, or `$psise.Options.ShowLineNumbers = $false`
- Cutting or copying the current line to the clipboard without highlighting it first
- Running the current script line (using F8) without highlighting it first

# The New Help Window

When you type a cmdlet name and press F1, the new help window will pop up if there is a match. If there is no match, an error message will appear. If you press F1 when the caret is not placed inside a cmdlet name or wildcard, the old CHM window appears, with instructions on how to use the ISE (has not been updated yet).

The new help window has font zoom, a search feature with search term highlighting and next/previous buttons. It also comes with persistent settings: You can specify which sections to display, and the help window will remember these settings next time. Here’s an example of a help window (and its settings) which only displays the synopsis, syntax, and examples:




It can also display about topics: Type “operators” and press F1.

## Add-On Tools

ISE now supports add-on tools, which are WPF controls, added via the object model to either the vertical or the horizontal tools pane. Multiple add-on tools in a pane are displayed as a tabbed control. Here’s a quick example:

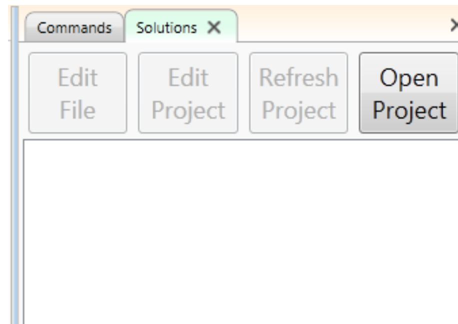
Download the ISESimpleSolution DLL to a path on your machine (say `$myPath`), and run the following:

```
[reflection.assembly]::LoadFile("$myPath\ISESimpleSolution.dll")

$psISE.CurrentPowerShellTab.VerticalAddOnTools.Add("Solutions",
[ISESimpleSolution.Solution], $true)
```

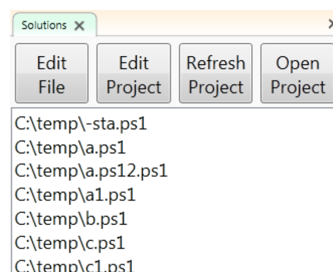
- The first parameter is the name of the tool.
- The second parameter is the type that implements the tool GUI. This has to be a UserControl WPF type that implements a particular interface. In this case this type is implemented in the DLL that was loaded. More on this type in the watch window example below or in the overview of the code for ISESimpleSolution.
- The third parameter is $true to show the add-on tool. $false would hide the add-on tool. Use the Add-Ons menu to show/hide individual tools.

Here’s what the Solutions add-on looks like:




Run the following in a folder containing scripts: `dir *.ps1 | %{$_.fullname} > project.txt`

Click in the "Open Project" button and select project.txt:




- Select a file and press "Edit File", or double-click a file to open it in the script pane
- Click "Edit Project" to edit the project.txt file
- If you change the text file and click refresh project, the list of projects will refresh

Notes on Add-On tools:

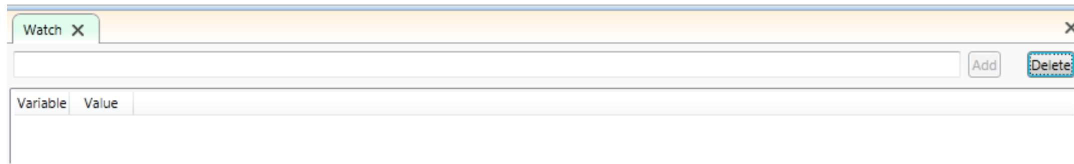
- To show/hide the vertical/horizontal pane: Use the "Show/Hide Vertical/Horizontal Add-On Tools Pane" option from the "Add-Ons" menu
- You can move an add-on tool from the vertical to the horizontal pane, using "Move Selected Vertical Add-On tool to Horizontal" (and vice versa) in the "Add-Ons" menu
- You can hide individual tools, but you can't actually close them. (The Close button might be misleading). To completely remove a tool, remove the code which loads it from your profile and restart the ISE

If you know a little WPF, ISESimpleSolution is an ideal place to start due to its simplicity:

- The GUI has to be a UserControl and implement the IAddOnToolHostObject interface
- The interface is very simple to implement: It is just a HostObject property set by ISE that gives the tool access to the root of the ISE object model ($psise in script)
- With this HostObject property you can interact with the ISE, as a typical Add-On to ISE would do. In this case, the tool just opens files in the current runspace with the following C# code: this.hostObject.CurrentPowerShellTab.Files.Add(fileName);
  This is equivalent to this script: `$psISE.CurrentPowerShellTab.Files.Add($fileName)`

The Watch Window Example:

Watchwindow.ps1 is a more complex Add-On tool.  When run:




If you type $a and click "Add" you will be watching $a. If the value changes inside or outside the debugger, you will see the new value displayed.

Notes:

- The watch window monitors changes in the CanInvoke property of the ISE Object Model in order to know when the list needs to be refreshed.
- The watch window uses the InvokeSynchronous method to evaluate variables.

## Restart Manager and Auto-Save

The ISE now auto-saves your open scripts every 2 minutes.  To change the auto-save interval, use: `$psise.Options.AutoSaveMinuteInterval`

If the ISE crashes or if the Operating System reboots, the ISE (upon starting) will recover the scripts which were open in the last session, even if they were unsaved.

## Command-Line Switches

Powerhsell_ise.exe now has the following new command-line switches:

- –File: Starts the ISE with the file(s) given
- –NoProfile: Starts the ISE without running $profile
- –Help: Displays a help window
- –mta: Starts the ISE in multi-threaded apartment mode.  (The default is STA)

## Other ISE Features

- MRU: A list of the most recently used files is available in the File menu.  The default is 10 files, but you can specify any number from 0 to 32 using `$psise.Options.MruCount`
- Persistent settings: All object model settings are now saved in AppData, so you don't have to keep them in your profile anymore.  Just have a settings script that you run only once.
- Update Help: A convenient way to update help content for beginners, located in the Help menu (must be run in an elevated ISE session)

- Convert text to uppercase/lowercase: This is actually a Win7 feature, but not a lot of people know about it, so here it is: Use Ctrl+U to convert selected text to lowercase, and use Ctrl+Shift+U to convert selected text to uppercase.

# Show-Command

Show-Command is a cool new feature which allows beginners to compose/run a cmdlet (or function) by filling out a "form", allowing them to use the graphical environment they're comfortable with, to run PowerShell cmdlets. Some will continue using PowerShell this way, and others may use Show-Command as a stepping stone to the command-line. Show-Command also enables advanced scripters to create a quick PowerShell-based GUI.

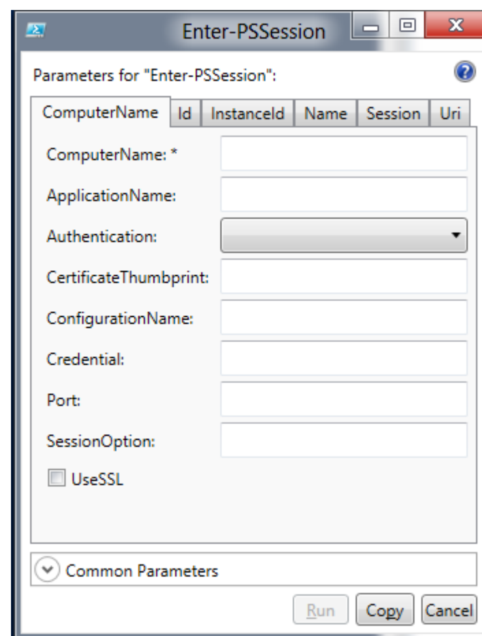
It is a stand-alone cmdlet which can be invoked from the classic console or from the ISE. There is also a built-in version of Show-Command in the ISE (as an add-on tool).

## Running Show-Command from the Command Line

There are 2 ways to invoke Show-Command from a PowerShell command line (classic console or ISE):

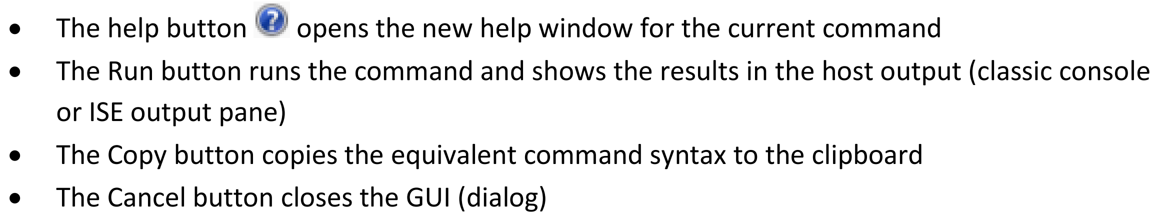
- By specifying a cmdlet name as a parameter, such as `Show-Command Enter-PSSession`. Alternatively, you can also type Enter-PSSession and press Ctrl+F1 (only in the ISE).
- Without a cmdlet name

Example: `Show-Command Enter-PSSession`



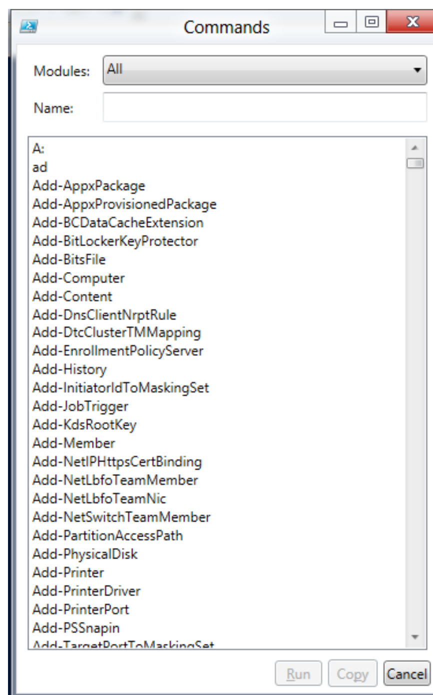

The GUI contains the parameter sets (each in a tab), along with the individual parameters. Mandatory parameters are marked with an asterisk (*), and the Run button won't be enabled until

all mandatory fields (for the current parameter set) have been filled out.  Common parameters are collapsed by default, but can be expanded and specified.

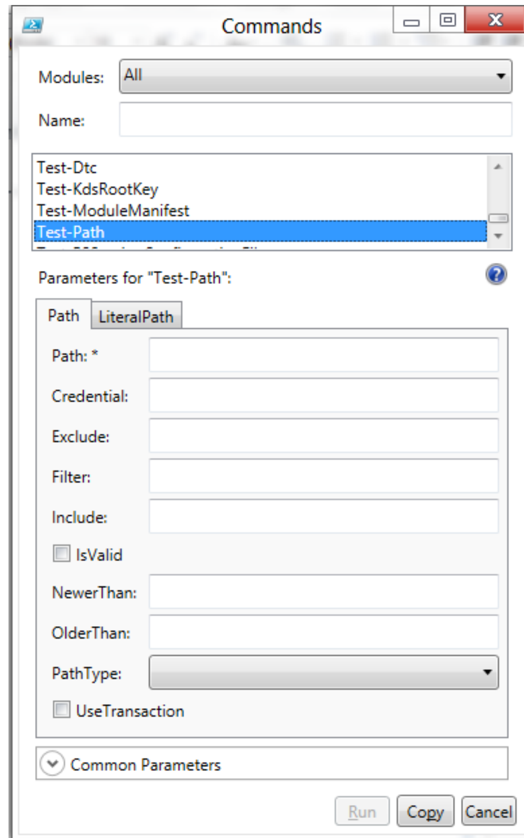


- The help button opens the new help window for the current command
- The Run button runs the command and shows the results in the host output (classic console or ISE output pane)
- The Copy button copies the equivalent command syntax to the clipboard
- The Cancel button closes the GUI (dialog)

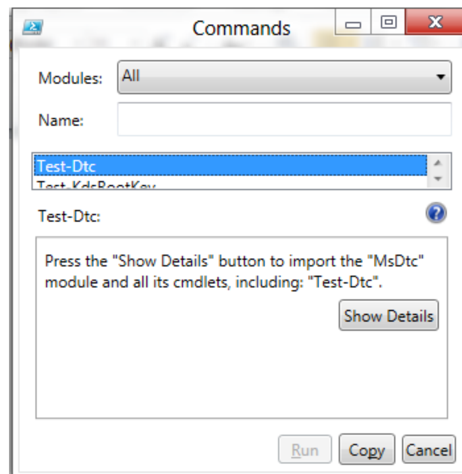
When you run Show-Command without a cmdlet name, all cmdlets will appear in the list.  When you hover over a cmdlet, a tooltip will display more information about the cmdlet, including the module it resides in: `Show-Command`




In addition to the controls we discussed previously, the Show-Command window has a text box to allow search by name, as well as a Modules drop-down list, to filter only cmdlets which are in a specific module.  Once a cmdlet is selected, its parameters and parameter sets will appear like before:




If you select a cmdlet whose module has not been imported yet, you see a button to import it labeled “Show Details”:

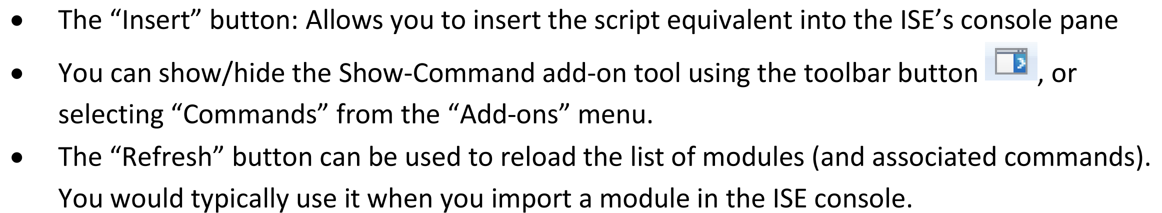



Other parameters for Show-Command include:

- –Height and –Width (of the Show-Command window in pixels)
- –NoCommonParameter: When specified, won’t display the common parameters section
- –NoErrorPopup: Won’t additionally display runtime errors as a pop-up message. (Note that errors are always displayed in the output anyways)

## Show-Command in the ISE

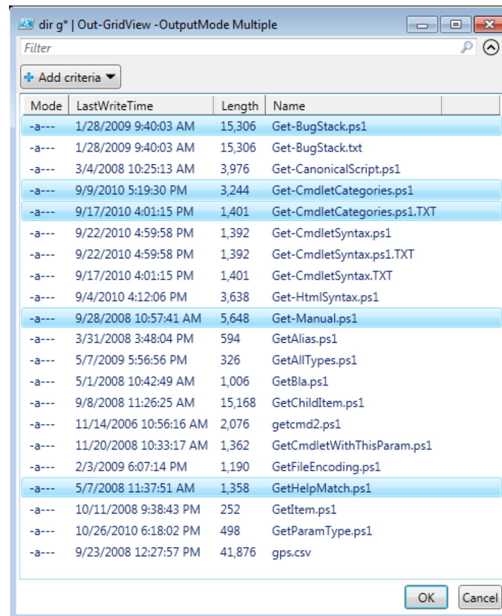
The Show-Command add-on tool in the ISE behaves almost identically to the stand-alone version described above, with a few minor differences:

- The “Insert” button: Allows you to insert the script equivalent into the ISE’s console pane
- You can show/hide the Show-Command add-on tool using the toolbar button , or selecting “Commands” from the “Add-ons” menu.
- The “Refresh” button can be used to reload the list of modules (and associated commands). You would typically use it when you import a module in the ISE console.

# Out-GridView

The new Out-GridView parameters allow you to select items from the list, and press a button to "pass through" only the objects which have been selected.

Here's a quick example: `dir g* | Out-GridView -OutputMode Multiple`




When you press "OK", only the selected objects are emitted:

```
Mode                LastWriteTime     Length Name
----                -------------     ------ ----
-a---         1/28/2009   9:40 AM      15306 Get-BugStack.ps1
-a---          9/9/2010   5:19 PM       3244 Get-CmdletCategories.ps1
-a---         9/17/2010   4:01 PM       1401 Get-CmdletCategories.ps1.TXT
-a---         9/28/2008  10:57 AM       5648 Get-Manual.ps1
-a---          5/7/2008  11:37 AM       1358 GetHelpMatch.ps1
```

Here are the new parameters at a glance:

- –OutputMode
  - None: No passthru at all (equivalent to the old Win7 behavior)
  - Single: Only one object can be selected for passthru
  - Multiple: Any number of objects can be selected for passthru
- –Passthru: Same as specifying –OutputMode Multiple
- –Block: Blocks the pipeline until the Out-GridView window is closed

The –Block parameter is especially useful when creating an application shortcut such as:

```
powershell.exe -command "Show-Command Get-Process |
Out-GridView -Block"
```

> Without –Block, PowerShell would close as soon as the results are in Out-GridView. When specifying –Block, PowerShell will only close when Out-GridView is closed. A shortcut like this enables an IT Pro to run a GUI which uses PowerShell, display results in another GUI, keeping the not-so-advanced IT Pro from using the command line.

Note that copying multiple lines to the clipboard (for pasting into Excel for example), is still supported.