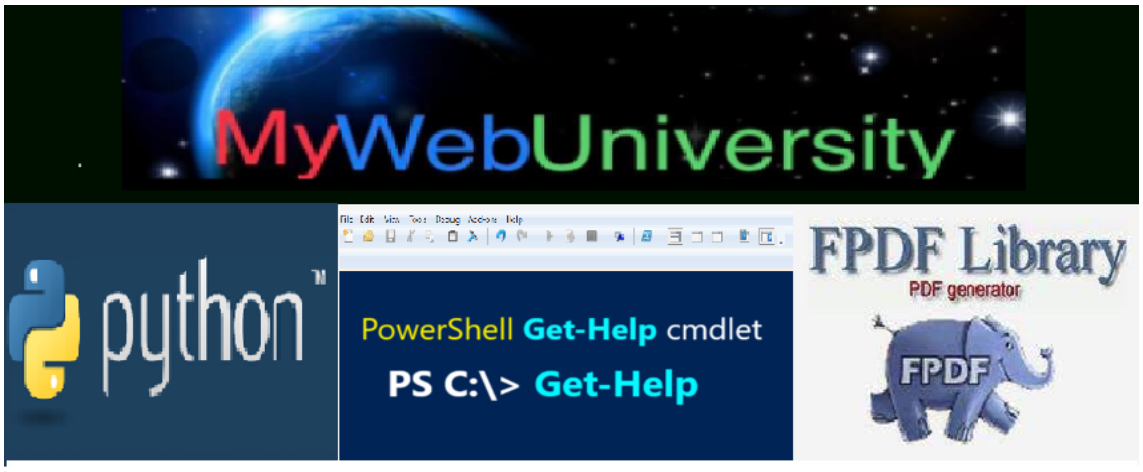

Full credit is given to all the above companies including the Operating System that this PDF file was generated!

Windows PowerShell Get-Help on Cmdlet 'Get-FileHash'

PS:\>Get-HELP Get-FileHash -Full

ABOUT_USING

Short description

Allows you to specify namespaces to use in the session.

Long description

The using statement allows you to specify namespaces to use in the session.
Adding namespaces simplifies usage of .NET classes and members and allows
you to import classes from script modules and assemblies.

The using statements must come before any other statements in a script or
module. No uncommented statement can precede it, including parameters.

The using statement must not contain any variables.

The using statement isn't the same as the using: scope modifier for variables. For more information, see about_Remote_Variables.

Namespace syntax

To resolve types from a .NET namespace:

```
using namespace <.NET-namespace>
```

Specifying a namespace makes it easier to reference types by their short names.

Example - Add namespaces for typename resolution

The following script gets the cryptographic hash for the "Hello World" string.

Note how the using namespace System.Text and using namespace System.IO simplify the references to [UnicodeEncoding] in System.Text and [Stream] and [MemoryStream] in System.IO.

```
using namespace System.Text
using namespace System.IO

[string]$string = "Hello World"
## Valid values are "SHA1", "SHA256", "SHA384", "SHA512", "MD5"
[string]$algorithm = "SHA256"

[byte[]]$stringBytes = [UnicodeEncoding]::Unicode.GetBytes($string)

[Stream]$memoryStream = [MemoryStream]::new($stringBytes)
$getFileHashSplat = @{
```

```
    InputStream = $memoryStream
    Algorithm   = $algorithm
}
$hashFromStream = Get-FileHash @getFileHashSplat
$hashFromStream.Hash.ToString()
```

Module syntax

To load classes and enumerations from a PowerShell module:

```
using module <module-name>
```

The value of <module-name> can be a module name, a full module specification, or a path to a module file.

When <module-name> is a path, the path can be fully qualified or relative. A relative path resolves relative to the script that has the using statement.

When <module-name> is a name or module specification, PowerShell searches the PSMODULEPATH for the specified module.

A module specification is a hashtable that has the following keys.

- ModuleName - REQUIRED Specifies the module name.
- GUID - OPTIONAL Specifies the GUID of the module.
- It's also REQUIRED to specify at least one of the three below keys.
  - ModuleVersion - Specifies a minimum acceptable version of the module.
  - MaximumVersion - Specifies the maximum acceptable version of the module.
  - RequiredVersion - Specifies an exact, required version of the

module. This can't be used with the other Version keys.

Import-Module and the #requires statement only import the module functions, aliases, and variables, as defined by the module. Classes and enumerations aren't imported.

The using module statement imports classes and enumerations from the root module (ModuleToProcess) of a script module or binary module. It doesn't consistently import classes or enumerations defined in nested modules or in scripts that are dot-sourced into the root module. Define classes and enumerations that you want to be available to users outside of the module directly in the root module.

During development of a script module, it's common to make changes to the code then load the new version of the module using Import-Module with the FORCE parameter. This works for changes to functions in the root module only. Import-Module doesn't reload any nested modules. Also, there's no way to load any updated classes or enumerations.

To ensure that you're running the latest version, you must start a new session. Classes and enumerations defined in PowerShell and imported with a using statement can't be unloaded.

Example - Load classes from a script module

In this example, a PowerShell script module named CARDGAMES defines the following classes:

- DECK
- CARD

Import-Module and the #requires statement only import the module functions, aliases, and variables, as defined by the module. Classes aren't imported.

The using module command imports the module and also loads the class definitions.

```
using module CardGames

[Deck]$deck = [Deck]::new()
$deck.Shuffle()
[Card[]]$hand1 = $deck.Deal(5)
[Card[]]$hand2 = $deck.Deal(5)
[Card[]]$hand3 = $deck.Deal(5)
```

Assembly syntax

The following syntax loads .NET types from an assembly into a script at the beginning of execution. You must use a fully-qualified path to the assembly file.

```
using assembly <.NET-assembly-path>
```

The using assembly statement is similar to using the Add-Type cmdlet. However, the Add-Type cmdlet adds the type at the time that Add-Type is executed, rather than at the start of execution of the script. For more information, see Add-Type.

Example - Load types from an assembly

This example loads an assembly so that its classes can be used when processing data. The following script converts data into a YAML format.

```
using assembly './YamlDotNet.dll'
using namespace YamlDotNet
```

```
$yamlSerializer = [Serialization.Serializer]::new()

$info = [ordered]@{
  Inventory = @(
    @{ Name = 'Apples' ; Count = 1234 }
    @{ Name = 'Bagels' ; Count = 5678 }
  )
  CheckedAt = [datetime]'2023-01-01T01:01:01'
}

$yamlSerializer.Serialize($info)

Inventory:
- Name: Apples
  Count: 1234
- Name: Bagels
  Count: 5678
CheckedAt: 2023-01-01T01:01:01.0000000
```