



@PoSh

🕒 Created	@January 14, 2025 8:57 AM
📁 Class	Scripting
📅 Date	@January 14, 2025

[Powershell Kata](#)

[PowerShell Fundamentals](#)

[What is PowerShell?](#)

[PowerShell Features](#)

[How is PowerShell Different?](#)

[Cmdlets in Detail](#)

[Check on Learning](#)

[Commandlets](#)

[Commandlets: Verb-Noun Naming Convention](#)

[PowerShell Pipeline](#)

[Parameters and Arguments](#)

[Check on Learning](#)

[PowerShell Help System](#)

[Get-Help](#)

[Parameters](#)

[Parameters in Detail](#)

[Parameter Sets](#)

[Get-Command](#)

[Get-ChildItem](#)

[Wildcard *](#)

[Get-Member](#)

[Check on Learning](#)

[PowerShell Aliases](#)

[Verifying Aliases](#)

[Check on Learning](#)

[Variables](#)

[User-Created Variables](#)

[Automatic Variables](#)

[Preference Variables](#)

[Check on Learning](#)

[Arrays](#)

[Arrays Example](#)

[Array Sub-Expression](#)

[Hash Tables](#)

[Hash Tables Example](#)

[Check on Learning](#)

[PowerShell Features: Profiles](#)

[PowerShell Profile Files](#)

[Editing a Profile](#)

[Creating a Profile](#)

[Editing \(Adding to\) the Profile](#)

[Check on Learning](#)

[PowerShell Objects](#)

[PowerShell Objects](#)

[PowerShell Pipeline](#)

[PowerShell One-Liner](#)

[Breaking Down the One-Liner](#)

[PowerShell Objects](#)

[Discovering Object Information](#)

[PowerShell Objects](#)

[PowerShell Objects: Methods and Properties](#)

[PowerShell Conditionals](#)

[If Statement](#)

[Else Statement](#)

[Elseif Statement](#)

[Comparison Operators](#)

[NOT \(! \) Operator](#)

[Other Operators](#)

[For Loop](#)

[ForEach Loop](#)

[PowerShell Scripts](#)

[Execution Policy](#)

[Execution Policy Documentation](#)

[PowerShell Functions](#)

[PowerShell Functions](#)

[Creating a PowerShell Function](#)

[Advanced Function Example](#)

[Apply Your Learning](#)

[String Manipulation in PowerShell](#)

[String Manipulation](#)

[Select-Object](#)

[ExpandProperty](#)

[Replace Operator](#)

[Join Operator](#)

[Split Method](#)

[Regular Expressions \(Regex\)](#)

[Match and Pattern](#)

[Repetition Quantifiers](#)

[Match and Pattern](#)

[Passing Matches to a ForEach Loop](#)

[Regex Example: Pentesting Guide](#)

[Anchors](#)

[PowerShell Scripts](#)

[Search and Replace Strings in a File](#)

[Final Challenge Problem](#)

[Solution Algorithms for PE Problems](#)

[Count the number of occurrences of a pattern in a file:](#)

[Count the number of Unique occurrences of a pattern in a file:](#)

[Which of the following is not a \[member\] available in \[cmdlet\] -type questions](#)

[Composable Resources](#)

[RegEx Patterns](#)

[1. Email Addresses](#)

[2. Dates \(MM/DD/YYYY or DD-MM-YYYY\)](#)

[3. Times \(HH:MM, 24-Hour Format\)](#)

[4. URLs](#)

[5. Social Security Numbers \(SSNs\)](#)

[6. MAC Addresses](#)

[7. Credit Card Numbers](#)

[8. US ZIP Codes](#)

[9. Hexadecimal Numbers](#)

[10. Floating-Point Numbers](#)

[11. RegEx for an Area Code Optionally Enclosed in Parentheses](#)

[12. RegEx for a Telephone Number](#)

13. RegEx for an IP Address

[RegEx Cheat Sheet:](#)

[Show Data-type](#)

Powershell Kata

```
get-<something> | where-object | where | ? {<property> -eq | -like <something>} | select-object -property | -expandproperty <property(ties)>
```

PowerShell Fundamentals

What is PowerShell?

PowerShell is a **cross-platform automation solution** made up of:

- A **command-line shell**.
- A **scripting language**.
- A **configuration management framework**.

PowerShell runs on Windows, Linux, and macOS.

As a scripting language:

- PowerShell is commonly used for automating the management of systems.
- It is also used to build, test, and deploy solutions, often in CI/CD environments.

CI/CD:

- Stands for Continuous Integration/Continuous Delivery or Deployment.
- Bridges development and operations by automating building, testing, and deployment.

Technical foundation:

- PowerShell is built on the `.NET Common Language Runtime (CLR)`.
- All inputs and outputs in PowerShell are `.NET objects`.

Objects in PowerShell:

- An object is an **enriched data container** representing data in an organized structure.
- **Methods**: Actions that can be taken by an object.



`$_` is short for `PSItem`. It is a placeholder for items in the pipeline. It references the output of the previous command and is used after a pipe.

PowerShell Features

PowerShell shares some features with traditional shells:

- **Built-in help system**: Provides information about commands and integrates with online help articles. `Get-Help`
 - use `Get-Help` followed by the command you would like help with to see a help page similar to Linux man pages.
 - You can search the help page using `-ShowWindow` followed by the search term.
 - At the bottom of the Help page you can see examples of command usage under `REMARKS`
 - you can also use `-examples` following `Get-Help` and the command
- **Pipeline**: Allows running many commands sequentially.
- **Aliases**: Alternate names used to run commands.
 - `Get-Alias` will show you aliases for common Linux commands.



Important RegEx:

`[]` Range

`{ }` Quantifier

`()` Group

How is PowerShell Different?

- **Object-oriented:** Operates on objects rather than text (bash outputs text).
 - A PowerShell object is made up of three types of data:
 - the object **type (ObjectType)**,
 - its **properties**,
 - and its **methods**.
 - No need to spend time formatting output and extracting data as objects retain their properties.
- **Cmdlets:** Commands in PowerShell are called `cmdlets` (pronounced "commandlets").
 - Cmdlets are built on a common runtime rather than being separate executables.

Cmdlets in Detail

- Cmdlets typically **take object input** and **return objects**.



All inputs and outputs are **.NET objects**.

- Core cmdlets in PowerShell are built in `.NET Core` and are open source.
- You can build custom cmdlets in `.NET Core` or PowerShell.

Types of commands in PowerShell:

- Native executables.
- Cmdlets.
- Functions.
- Scripts.
- Aliases.

Check on Learning

- What are some features PowerShell shares with traditional shells?
 - Pipelines
 - Help-Pages
 - Aliases
 - What are commands in PowerShell called?
 - cmdlets
-

Commandlets

A `cmdlet` is a lightweight command used in the PowerShell environment.

Key differences from traditional commands:

- Cmdlets are instances of `.NET classes`; they are not stand-alone executables.
 - Cmdlets process input objects from the pipeline rather than streams of text.
 - Cmdlets deliver objects as output to the pipeline.
-

Commandlets: Verb-Noun Naming Convention

- Cmdlets follow a **verb/noun naming convention**.
- Example: `Get-ChildItem`.
 - References the directory that you are already in and delivers all items within it. (similar to `ls` in Linux)

Functionality:

- Lists or returns items in one or more specified locations.
- If items are in a container, the command retrieves the items inside the container — child items.

Common Verbs include:

- Get-*

- Start-*
- Stop-*
- Read-*
- Write-*
- New-*
- Out-*

PowerShell Pipeline

- A pipe (|) takes the output of the first part of the pipeline and uses it as the input for the next part.
- Example: Using the `Select-Object` cmdlet to display only the `DisplayName` property.

```
Steve: C:\Users\student >>>get-service | Select-Object Name
Name
----
AarSvc_4639e
AJRouter
ALG
```

Parameters and Arguments

- **Cmdlets:** Most cmdlets support parameters as part of their input mechanism. These are similar to command *options* in Linux.
- Parameters can be added:
 - At the command line.
 - Passed through the pipeline as output from a previous cmdlet.

Arguments:

- Specify the input a cmdlet accepts.

- Control how the cmdlet operates.
- Define what data, if any, the cmdlet outputs.

Example:

To filter services that are running:

```
Get-Service | Where-Object {$_.Status -eq "Running"}
```

Status	Name	DisplayName
Stopped	AarSvc_28ac7e8a	Agent Activation Runtime_28ac7e8a
Stopped	AJRouter	AllJoyn Router Service
Stopped	ALG	Application Layer Gateway Service
Stopped	AppIDSvc	Application Identity
Running	Appinfo	Application Information
Running	AppMgmt	Application Management
Stopped	AppReadiness	App Readiness
Stopped	AppVClient	Microsoft App-V Client
Stopped	AppXSvc	AppX Deployment Service (AppXSVC)
Stopped	AssignedAccessM...	AssignedAccessManager Service
Running	AudioEndpointBu...	Windows Audio Endpoint Builder
Running	Audiosrv	Windows Audio
Stopped	autotimesvc	Cellular Time
Stopped	AxInstSV	ActiveX Installer (AxInstSV)
Stopped	BcastDVRUserSer...	GameDVR and Broadcast User Service...
Stopped	BDESVC	BitLocker Drive Encryption Service
Running	BFE	Base Filtering Engine
Stopped	BITS	Background Intelligent Transfer Se...
Stopped	BluetoothUserSe...	Bluetooth User Support Service_28a...
Running	BrokerInfrastru...	Background Tasks Infrastructure Se...

Check on Learning

1. What type of naming convention do cmdlets use?
 - Verb-Noun

2. Which statement is true about how cmdlets differ from commands?

- a. Cmdlets are stand-alone executables.
- b. Cmdlets are instances of .NET classes.
- c. Both a & b.

PowerShell Help System

- Most shells include a help system for learning about commands, their functions, and supported parameters.

PowerShell Help Cmdlets:

- `Get-Help`
- `Get-Command`
- `Get-Member`

Get-Help

- `Get-Help` is a multipurpose cmdlet that helps you learn how to use commands.
- It searches:
 1. For wildcard matches of command names based on provided input.
 2. Through help topics if no direct match is found.

Example:

```
Get-Help -Name Get-Help
```

We are calling `Get-Help` on the cmdlet named `Get-Help`. Similar to `man man` in Linux.

```
Windows PowerShell
PS C:\Users\student\Downloads> get-help -name get-help


NAME
    Get-Help

SYNOPSIS
    Displays information about PowerShell commands and concepts.

SYNTAX
    Get-Help [[-Name] <System.String>] [-Category {Alias | Cmdlet | Provider | General | FAQ | Glossary | HelpFile | ScriptCommand | Function | Filter | ExternalScript | All | DefaultHelp | Workflow | DscResource | Class | Configuration}] [-Component <System.String[]>] [-Detailed] [-Functionality <System.String[]>] [-Path <System.String>] [-Role <System.String[]>] [<CommonParameters>]
```

Get-Help (Microsoft.PowerShell.Core) - PowerShell

The Get-Help cmdlet displays information about PowerShell concepts and commands, including cmdlets, functions, Common Information Model (CIM) commands, workflows,

 https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.core/get-help?view=powershell-7.4&viewFallbackFrom=powershell-7.3&WT.mc_id=ps-gethelp



Parameters

Run the example command on your computer and review the output. Information is grouped into:

- **NAME**
- **SYNOPSIS**
- **SYNTAX**
- **DESCRIPTION**

- **RELATED LINKS**
- **REMARKS**

Help topics can contain significant amounts of information—this is just part of the output.

Parameters in Detail

- **Parameter:** A way to provide input to a command.
- `Get-Help` has many parameters to return the entire help topic or subsets of it.

Parameter Sets:

- Multiple blocks in the syntax section represent different parameter sets.
 - Each set has at least one unique parameter.
 - Example: The `Full` and `Detailed` parameters are in different parameter sets and cannot be used together.
-

Parameter Sets

- Parameter sets are **mutually exclusive**.
- Once a parameter unique to one set is used, only other parameters in that set can be used.

Examples of Parameters in Different Sets:

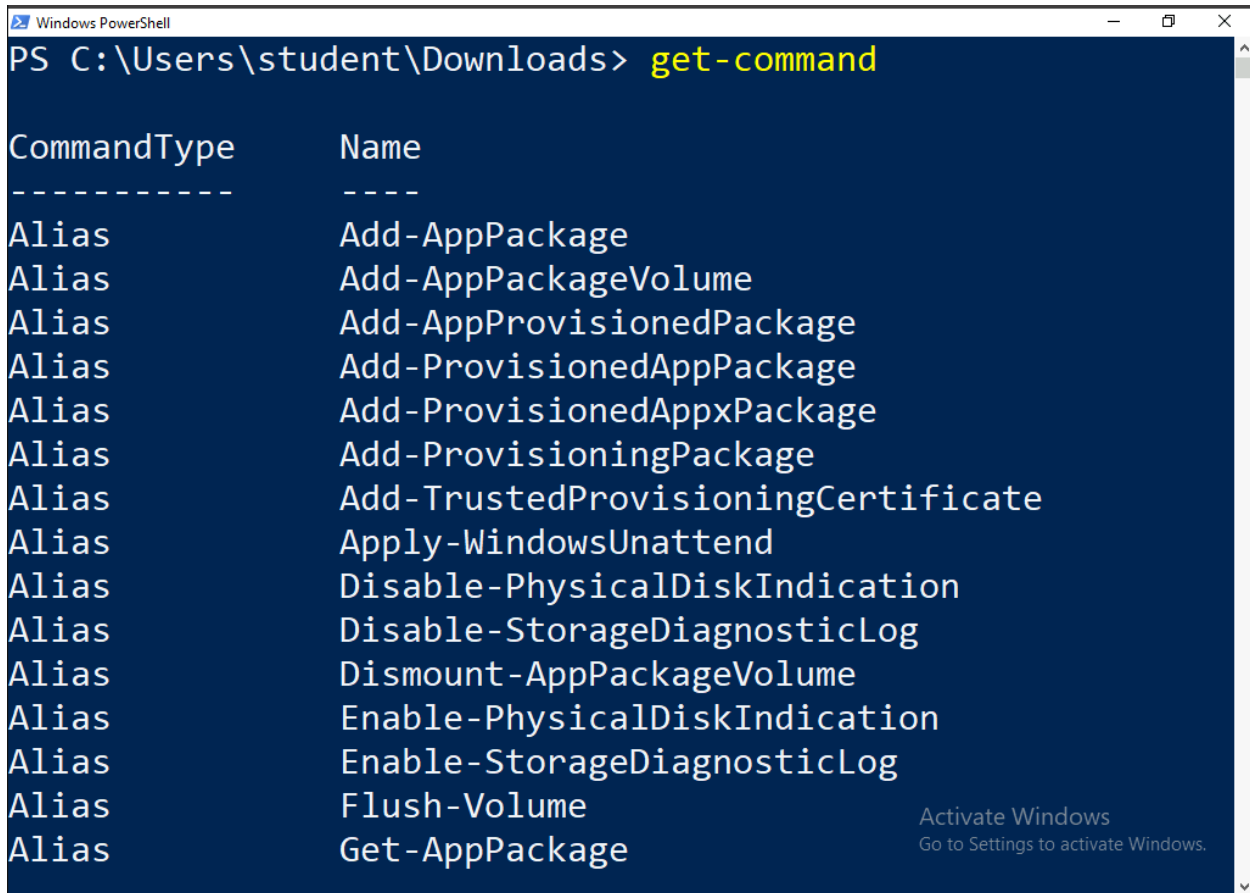
- `Full`
 - `Detailed`
 - `Examples`
 - `Online`
 - `Parameter`
 - `ShowWindow`
-

Get-Command

- `Get-Command` retrieves all commands installed on the computer.
- Includes: cmdlets, aliases, functions, filters, scripts, and applications.

Syntax:

```
Get-Command
```



```

Windows PowerShell
PS C:\Users\student\Downloads> get-command

CommandType      Name
-----
Alias             Add-AppPackage
Alias             Add-AppPackageVolume
Alias             Add-AppProvisionedPackage
Alias             Add-ProvisionedAppPackage
Alias             Add-ProvisionedAppxPackage
Alias             Add-ProvisioningPackage
Alias             Add-TrustedProvisioningCertificate
Alias             Apply-WindowsUnattend
Alias             Disable-PhysicalDiskIndication
Alias             Disable-StorageDiagnosticLog
Alias             Dismount-AppPackageVolume
Alias             Enable-PhysicalDiskIndication
Alias             Enable-StorageDiagnosticLog
Alias             Flush-Volume
Alias             Get-AppPackage
  
```

Get-ChildItem

- Retrieves the parameter sets of the `Get-ChildItem` command.

Syntax:

```
Get-Command -Name Get-ChildItem -Syntax
```

You can replace `Get-ChildItem` with any command to retrieve its syntax.

We are calling `Get-Command` on the cmdlet named `Get-ChildItem` to see its accepted syntax.

```
Windows PowerShell
PS C:\Users\student\Downloads> get-command get-childitem -syntax

Get-ChildItem [[-Path] <string[]>] [[-Filter] <string>] [-Include <string[]>] [-Exclude <string[]>] [-Recurse] [-Depth <uint32>] [-Force] [-Name] [-UseTransaction] [-Attributes <FlagsExpression[FileAttributes]>] [-Directory] [-File] [-Hidden] [-ReadOnly] [-System] [<CommonParameters>]

Get-ChildItem [[-Filter] <string>] -LiteralPath <string[]> [-Include <string[]>] [-Exclude <string[]>] [-Recurse] [-Depth <uint32>] [-Force] [-Name] [-UseTransaction] [-Attributes <FlagsExpression[FileAttributes]>] [-Directory] [-File] [-Hidden] [-ReadOnly] [-System] [<CommonParameters>]
```

Wildcard *

- The wildcard character `*` matches any and all string filters.
- **Syntax:**

We are calling

`Get-Command` and filtering for commands that start with `Get`.

```
Get-Command Get*
```

Get-Member

- The `Get-Member` cmdlet retrieves the **members**, **properties**, and **methods** of objects.
 - It will show you all of the possible properties of a particular command as well as all the methods you can call on it.

- **Syntax:**

Takes

`Get-Process` and pipes it into `Get-Member` to display all the methods and properties of that object.

```
Get-Process | Get-Member
```

- **Methods:** Things we can do with these objects.
- **Properties:** Things these objects are.

Check on Learning

1. What does the `Get-Member` cmdlet do?
 - a. Tells you about a command! Retrieves the **members, properties, and methods**.
2. True or False: Parameter sets are not mutually exclusive.
 - a. False! If you were to run `Examples` and `Full` it would not know what to do!

PowerShell Aliases

- An **alias** is an alternate name for a cmdlet, function, executable file, or script.
- PowerShell includes built-in aliases (e.g., `cls` for `Clear-Host` or `ls` for `Get-ChildItem`).
- Users can create their own aliases for the current session or save them to the PowerShell profile.

To create an alias:

```
Set-Alias -Name list -Value Get-ChildItem
```

Verifying Aliases

- Use the `Get-Alias` cmdlet to verify that an alias was created.
- **Syntax:**

This checks that the alias

`list` functions exactly like `Get-ChildItem`.

```
Get-Alias -Name list
```

To find all aliases for a command:

```
Get-Alias -Definition Get-ChildItem
```

Check on Learning

1. What is the full command to create an alias?
 - a. `Set-Alias -name <name> -value <cmdlet>`
2. True or False: The `Get-Alias` cmdlet can verify an alias was created.
 - a. True!

Variables

- A **variable** is a unit of memory where values are stored.
- A variable stores **one** item.
- In PowerShell, variables are represented by text strings starting with `$` (e.g., `$a`, `$process`, `$my_var`).
- Variable names are not case-sensitive and can include spaces or special characters.
- PowerShell will make an educated guess at the data type of a variable if you give it an int, float, etc.

User-Created Variables

- Created and maintained by the user.
- **By default, user-created variables exist only while the PowerShell window is open.**
- To save variables:
 - Add them to your PowerShell profile.
 - Use `global`, `script`, or `local` scope in scripts.

```
#User-created variable  
$my_var = 1  
echo $my_var  
  
$my_var += 1  
echo $my_var
```

Automatic Variables

- **Automatic variables** store the state of PowerShell.
- Created and updated automatically by PowerShell; users cannot modify their values.
- PowerShell changes their values as required to maintain their accuracy.
- **Example:**
 - `$PSHOME`: Stores the path to the PowerShell installation directory.

```
#Automatic variable  
echo $PSVersionTable  
echo $PSHome
```

Preference Variables

- **Preference variables** store user preferences for PowerShell.
- Created by PowerShell but users can modify their values.

- **Example:**

- `$MaximumHistoryCount` : Determines the maximum number of entries in the session history

```
#Preference variable
echo $MaximumAliasCount
echo $MaximumHistoryCount
```

Check on Learning

1. List some of the different types of variables discussed in PowerShell.
 - a. User Created
 - b. Automatic
 - c. Preference
2. True or False: Users can change preference variables.
 - a. True!

Arrays

- An **array** is a data structure designed to store a collection of items.
- Items in an array can be of the same or different types.
- Arrays are organized by **zero-based index**.
- Remember how a variable stores **one** item? An array is a kind of variable that stores many values.

To create and initialize an array:

- Assign multiple values to a variable.
- Delimit the values with a comma and separate them from the variable name using the assignment operator (`=`).

```
$A = 1,5,10,15,10,"squirrel"  
#In this example we are taking the values 22,5,10,8,12,9 and 80 and storing them into the array named $A  
  
echo $A
```

Arrays Example

- The data is associated in memory with the array named `$A`.
- Anytime we want to access this information, we can interact with the array.
- Example: The value `5` is stored at index `1`.

```
#We can access the value at a specific position by referring to its index.  
  
echo $A[1]
```

Array Sub-Expression

- The **array sub-expression operator** creates an array from the statements inside it.
- Whatever the statement inside the operator produces, it will be placed in an array, even if there is zero or one object.

Sub Expression Syntax

```
@( commands )
```

Sub Expression Syntax

```
$services = @(get-service | where {$_.Status -eq "Running"} | Select-Object Name )
```

Example:

An array named `$services` contains a collection of all service names with a status of `"Running"`.

Usage:

- We can now interact with service names as if they were a regular array.

```
echo $services[12]
```

- This organizes output from a cmdlet into an array for further data processing or analysis.
 - One of the properties of an array is length. This is simply how many objects are in an array.

Array Length:

- Access the number of objects in an array using the `.Length` method:

```
$array.Length
```

How would you access the last value in an array of unknown length?

```
echo $VarName[$_.length-1]
```

```
# OR, more simply...
```

```
echo $VarName[-1]
```

Hash Tables

- A **hash table** (also known as a dictionary or associative array) is a compact data structure that stores one or more **key/value pairs**.
- Example:
 - Keys: IP addresses.
 - Values: Computer names (or vice versa).

Key-Based Indexing:

- Unlike regular arrays, hash tables use **keys** to access values, not numerical positions.
-

Hash Tables Example

To create a hash table:

- Follow this syntax:

```
$phonebook = @{ Bob = "706-123-4567"; Alice = "803-123-4567"; Steve = "555-123-4567" }
```

```
$hashTable = @{  
    Key1 = "Value1";  
    Key2 = "Value2";  
    Key3 = "Value3"  
}
```

- Keys are associated with their corresponding values.
- Reference a specific key to access its value:

```
echo $phonebook["Bob"]  
echo $phonebook["Alice"]
```

Check on Learning

1. What is an array?
 - a. An **array** is a data structure designed to store a collection of items. Items may be alike or different. It is zero-indexed and comma delimited.
2. What are the values stored in an array delimited with?
 - a. Commas (,)!

PowerShell Features: Profiles

- A **PowerShell profile** customizes your environment and adds session-specific elements to every session you start. (Just like .bashrc in Linux! (Bash Run Configuration))

- Profiles are scripts that run when PowerShell starts.
- You can use profiles as logon scripts to configure the environment.

Elements you can add to a profile:

- Commands.
 - Aliases.
 - Functions.
 - Variables.
 - Snap-ins.
 - Modules.
 - PowerShell drives.
 - Other session-specific elements.
-

PowerShell Profile Files

- PowerShell supports several profiles for users and host programs.

Description	Path
All Users, All Hosts	Windows \$PSHOME\Profile.ps1 Linux /usr/local/microsoft/powershell/7/profile.ps1 macOS /usr/local/microsoft/powershell/7/profile.ps1
All Users, Current Host	Windows \$PSHOME\Microsoft.PowerShell_profile.ps1 Linux /usr/local/microsoft/powershell/7/Microsoft.PowerShell_profile.ps1 macOS /usr/local/microsoft/powershell/7/Microsoft.PowerShell_profile.ps1
Current User, All Hosts	Windows \$Home\Documents\PowerShell\Profile.ps1 Linux ~/.config/powershell/profile.ps1 macOS ~/.config/powershell/profile.ps1
Current user, Current Host	Windows \$Home\Documents\PowerShell\Microsoft.PowerShell_profile.ps1 Linux ~/.config/powershell/Microsoft.PowerShell_profile.ps1 macOS ~/.config/powershell/Microsoft.PowerShell_profile.ps1

- **Important:** PowerShell does not create profiles automatically; you need to create them.

Editing a Profile

- You can open any PowerShell profile in a text editor (e.g., Notepad).
- **Automatic Variable:** Use `$PROFILE` to reference the path of your PowerShell profile.

```
notepad $PROFILE
```

```
#To specify a specific profile.  
notepad $PROFILE.AllUsersAllHosts
```

Test the path of the \$PROFILE to see if it exists:

```
Test-Path $profile
```

Creating a Profile

- If `Test-Path $PROFILE` returns `false`, it means the file doesn't exist.
- To create the profile:

```
New-Item -Path $PROFILE -ItemType File -Force
```

- Test it again:

```
Test-Path $PROFILE
```

It should now return `true`.

To find the profile path:

```
$PROFILE
```

- Navigate to the path and open the file.
- In this file, you can make changes that will be applied every time you open PowerShell.

Editing (Adding to) the Profile

- To modify the profile, add desired commands, aliases, or configurations directly to the file.

- We are going to add the following to our profile:

```
function prompt{
    "CYBERMAN: $($ExecutionContext.SessionState.Path.CurrentLocation) $('= ')"
}
set-alias -name list -value get-childitem
```

- **After editing:**
 - Save the profile file.
 - Restart PowerShell to apply the changes.

Check on Learning

1. True or False: A PowerShell profile is a script that runs when PowerShell starts.
2. List two of the PowerShell profiles.

PowerShell Objects

PowerShell Objects

- PowerShell is an **object-oriented language and shell**.
- Unlike traditional shells (e.g., cmd, Bash) that focus on text (strings), PowerShell focuses on **objects**.
- Nearly everything in PowerShell is an object.
- **Key Advantage:** Leverage its object-oriented structure to accomplish complex goals efficiently.

PowerShell Pipeline

- A **pipeline** is a series of commands connected by the pipeline operator (`|`).

- **How it works:**

- Each pipeline operator sends the results of the preceding command to the next command.
- The output of one command becomes the input for the next command in the pipeline.

Example:

- Start an instance of `notepad.exe`.
- Use `Get-Process` to find the `notepad` process.
- Pipe the process object into `Stop-Process` to stop the notepad process.

```
notepad.exe  
Get-Process notepad | stop-process
```

```
Get-Process -Name notepad | Stop-Process
```

- The pipeline takes the output of the command on the left and pipes it as input to the command on the right.

PowerShell One-Liner

- A PowerShell **one-liner** is a continuous pipeline.
- **Note:** A one-liner doesn't have to be on a single physical line.

Example (a multi-line one-liner):

```
Get-Service |  
  Where-Object StartType -eq Automatic |  
  Select-Object -Property Name
```

```
Get-Service |  
Where-Object {$_.StartType -eq "Automatic"} |
```

```
Select-Object -Property Name
```

Breaking Down the One-Liner

```
Get-Service |  
    Where-Object StartType -eq Automatic |  
        Select-Object -Property Name
```

1. Step 1:

- `Get-Service` retrieves all services.
- Output is piped to the next command.

2. Step 2:

- `Where-Object` filters the collection of services.
- Selects services where the `StartType` property equals `"Automatic"`.

3. Step 3:

- `Select-Object` retrieves the `Name` property of the filtered services.
 - Since it is the last pipe, the output is displayed on the screen.
-

PowerShell Objects

- Objects have multiple types of information, referred to as **members**.
- **Members:**
 - Generic term for all information associated with an object.
 - Includes **properties**, **methods**, and more.
- An object can have multiple members, but each member belongs to a single object.

```
PS C:\Users\student\Downloads> get-process | get-member

TypeName: System.Diagnostics.Process

Name            MemberType      Definition
----            -
Handles         AliasProperty  Handles = Handlecount
Name            AliasProperty  Name = ProcessName
NPM             AliasProperty  NPM = NonpagedSystemMemoryS...
PM             AliasProperty  PM = PagedMemorySize64
SI             AliasProperty  SI = SessionId
VM             AliasProperty  VM = VirtualMemorySize64
WS             AliasProperty  WS = WorkingSet64
Disposed        Event          System.EventHandler Dispose...
ErrorDataReceived Event          System.Diagnostics.DataRece...
Exited          Event          System.EventHandler Exited(...
OutputDataReceived Event          System.Diagnostics.DataRece...
BeginErrorReadLine Method          void BeginErrorReadLine()
BeginOutputReadLine Method          void BeginOutputReadLine()
CancelErrorRead Method          void CancelErrorRead()
CancelOutputRead Method          void CancelOutputRead()
Close           Method          void Close()
CloseMainWindow Method          bool CloseMainWindow()
```

Discovering Object Information

- Use the `Get-Member` cmdlet to discover object information.
- **Example:**

```
Get-Service | Get-Member
```

```
PS C:\Users\sinpra> Get-service | Get-Member

TypeName: System.ServiceProcess.ServiceController

Name      MemberType Definition
-----
Name      AliasProperty Name = ServiceName
RequiredServices AliasProperty RequiredServices = ServicesDependedOn
Disposed  Event System.EventHandler Disposed(System.Object)
Close     Method void Close()
Continue  Method void Continue()
CreateObjRef Method System.Runtime.Remoting.ObjRef CreateObjRef()
Dispose   Method void Dispose(); void IDisposable.Dispose()
Equals    Method bool Equals(System.Object obj)
ExecuteCommand Method void ExecuteCommand(int command)
GetHashCode Method int GetHashCode()
GetLifetimeService Method System.Object GetLifetimeService()
GetType   Method Type GetType()
InitializeLifetimeService Method System.Object InitializeLifetimeService()
Pause     Method void Pause()
Refresh   Method void Refresh()
Start     Method void Start(); void Start(string[] args)
Stop      Method void Stop()
WaitForStatus Method void WaitForStatus(System.ServiceProcess.ServiceController)
CanPauseAndContinue Property bool CanPauseAndContinue {get;}
CanShutdown Property bool CanShutdown {get;}
CanStop   Property bool CanStop
Container Property System.ComponentModel.IContainer
DependentServices Property System.ServiceProcess.ServiceController
DisplayName Property string DisplayName {get;set;}
MachineName Property string MachineName {get;set;}
ServiceHandle Property System.Runtime.InteropServices.SafeHandle
ServiceName Property string ServiceName {get;set;}
ServicesDependedOn Property System.ServiceProcess.ServiceController
ServiceType Property System.ServiceProcess.ServiceType
Site      Property System.ComponentModel.ISite Site {get;set;}
Status    Property System.ServiceProcess.ServiceController
ToString  ScriptMethod System.Object ToString();
```

- This command reveals:
 - The type of object returned (`System.ServiceProcess.ServiceController`).
 - The available members of the object (e.g., properties, methods).

PowerShell Objects

- **Methods:** Actions that can be performed on an object.
 - Examples: `Pause` , `Stop` , `Start` (used to manage services).
- **Properties:** Attributes that describe an object.
 - Examples: `DisplayName` , `ServiceName` , `Status` .

```
Select Windows PowerShell
PS C:\Users\student\Downloads> get-service | get-member

TypeName: System.ServiceProcess.ServiceController

Name            MemberType      Definition
----            -
Name            AliasProperty  Name = ServiceName
RequiredServices AliasProperty  RequiredServices = ServicesDe...
Disposed        Event          System.EventHandler Disposed(...)
Close           Method         void Close()
Continue        Method         void Continue()
CreateObjRef     Method         System.Runtime.Remoting.ObjRe...
Dispose         Method         void Dispose(), void IDisposa...
Equals          Method         bool Equals(System.Object obj)
ExecuteCommand   Method         void ExecuteCommand(int command)
GetHashCode      Method         int GetHashCode()
GetLifetimeService Method       System.Object GetLifetimeServ...
GetType         Method         type GetType()
InitializeLifetimeService Method       System.Object InitializeLifet...
Pause           Method         void Pause()
Refresh         Method         void Refresh()
```

PowerShell Objects: Methods and Properties

- **Method Members:** Actions taken on an object.
 - Example: For a "Human" object, methods might include `Walk`, `Run`, `Breathe`.
- **Property Members:** Attributes that describe an object.
 - Example: A "Human" object might have properties like `Height`, `Weight`, `Age`.
- **To search for properties:**

Use the `Where-Object` cmdlet to filter objects based on their properties.

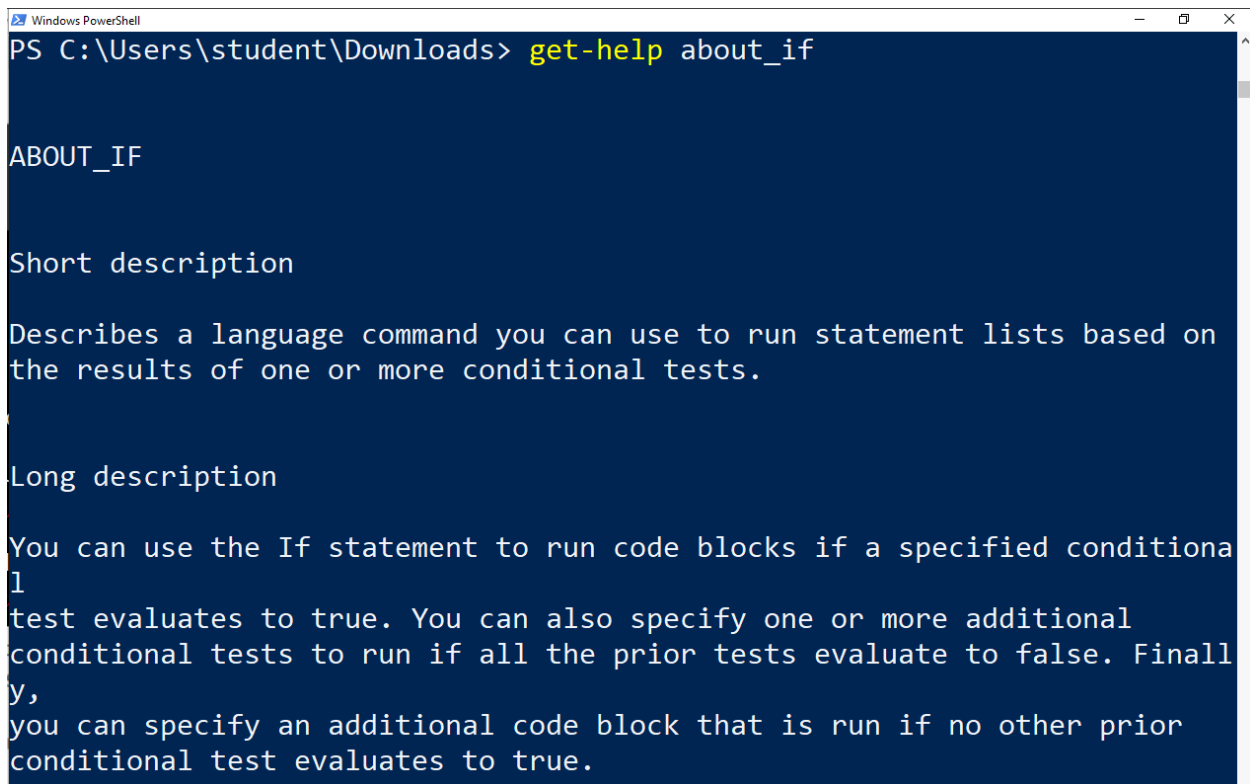
PowerShell Conditionals

- **Procedural Programming:**
 - Derived from imperative programming.
 - Based on the concept of the **procedure call** (a series of computational steps).

- All procedural languages, including PowerShell, require constructs for condition-based execution of instructions.

If Statement

- PowerShell, being procedural, includes built-in **conditionals** like the `If` statement.



```
Windows PowerShell
PS C:\Users\student\Downloads> get-help about_if

ABOUT_IF

Short description

Describes a language command you can use to run statement lists based on
the results of one or more conditional tests.

Long description

You can use the If statement to run code blocks if a specified conditiona
l
test evaluates to true. You can also specify one or more additional
conditional tests to run if all the prior tests evaluate to false. Finall
y,
you can specify an additional code block that is run if no other prior
conditional test evaluates to true.
```

- **How it works:**
 - Evaluates the condition inside parentheses.
 - If the condition is `$true`, executes the script block inside the braces `{ }`.
 - If the condition is `$false`, skips the script block.

```
$condition = $true
if ( $condition )
{
    Write-Output "The condition was true"
}
```

Example:

```
$x = 1
if ( $x -eq 1 )
{
    Write-Output "The condition was true"
}
```

Else Statement

- The `Else` statement does not accept any conditions.
- It runs **only if all prior conditions are** `$false`.
- Acts as the **"last resort" condition**.

```
$condition = $false
if ( $condition )
{
    Write-Output "The condition was true"
}
else
{
    Write-Output "The condition was false"
}
```

Elseif Statement

- Use the `ElseIf` statement to add additional conditions.
- You can include multiple `ElseIf` statements for different conditions.
- PowerShell evaluates the conditions **sequentially**.


```
$condition = $false
if ( $condition )
{
    Write-Output "The condition was true"
}
elseif (!$condition)
{
    Write-Output "The condition was false"
}
else
{
    Write-Output "I dont know how you got here."
}
```

Comparison Operators

- The `If` statement is often used to **compare two items**.
- **PowerShell Comparison Operators:**
 - Compare the value on the **left-hand side** with the value on the **right-hand side**.
 - Examples include:
 - `eq` (equals).
 - `ne` (not equals).
 - `lt` (less than).
 - `gt` (greater than).

```
Get-Help about_Comparison_Operators
```

NOT (`!`) Operator

- The `!` operator inverts a Boolean value:

- A `$false` statement becomes `$true`.
- A `$true` statement becomes `$false`.

Examples:

```
$condition = $false
if ( !$condition )
{
    Write-Output "The condition was true"
}
Write-Output (!$condition)
```

```
if (-not $condition) {
    Write-Output "Condition was false."
}
```

Other Operators

- PowerShell includes a variety of operators for tasks like arithmetic, logical comparisons, and more.
- Common examples:
 - **Logical Operators:** `and`, `or`, `not`.
 - **Arithmetic Operators:** `+`, `-`, `*`, `/`.
 - **Assignment Operators:** `=`, `+=`, `-=`.

eq - Equal

```
$value = 5
if ( 5 -eq $value )
{
    Write-Output "The value is $value"
}
```

ne - Not Equal

```
$value = 4
if ( 5 -ne $value )
{
    Write-Output "The value $value is not 5"
}
```

gt - Greater Than

```
$value = 4
if ( 5 -gt $value )
{
    Write-Output "The value 5 is greater than $value"
}
```

ge - Greater Than or Equal

```
$value = 5
if ( 5 -ge $value )
{
    Write-Output "The value 5 is greater than or equal to $value"
}
```

lt - Less Than

```
$value = 6
if ( 5 -lt $value )
{
    Write-Output "The value 5 is less than $value"
}
```

le - Less Than or Equal

```
$value = 5
if ( 5 -le $value )
{
    Write-Output "The value 5 is less than or equal to $value"
}
```

is - Is the same type

```
$value = "test"
if ( $value -is [string] )
{
    Write-Output "The value $value is a string"
}
```

For Loop

```
for (<Init>; <Condition>; <Repeat>)
{
    <Statement list>
}
```

- The `For` loop has four key components:
 1. **Init:** Commands executed before the loop begins.
 2. **Condition:** Evaluates to `$true` or `$false` each time the loop runs.
 - If `$true`, the commands in the loop's block are executed.
 - If `$false`, the loop ends.
 3. **Repeat:** Commands executed each time the loop repeats.
 4. **Statement:** Commands run each time the loop is entered or repeated, enclosed in braces `{ }`.

Example:

```
for ($i;$i -lt 15; $i++)  
{  
    Write-Output "Loop Number: $i"  
}  
$i = 0
```

```
for ($i = 0; $i -lt 5; $i++) {  
    Write-Output "Iteration $i"  
}
```

ForEach Loop

- A `foreach` loop iterates through a collection of objects and runs commands for each item.
- **Typical use case:** Traversing an array.
- **Key differences from `For` loop:**
 - `For`: Loops until a condition is met, potentially infinite.
 - `ForEach`: Loops a set number of times based on the collection's size.

Syntax:

```
foreach ($<item> in $<collection>)  
{  
    <statement list>  
}
```

```
foreach ($item in $collection) {  
    Write-Output "Processing $item"  
}
```

- The variable `$item` is local to the loop and named by the user.

PowerShell Scripts

- A **script** is a plain text file containing PowerShell commands with a `.ps1` file extension.
- **How to run a script:**
 - Type the script's path and file name.
 - Use parameters to submit data and set options.

Advantages of Scripts:

- Save commands for later use.
- Simplify sharing commands.
- Execute multiple commands with a single script file.

Additional Features:

- `#Requires` special comment.
- Use of parameters and data sections.
- Digital signing for security.
- Ability to write Help topics for scripts and functions.

Execution Policy

- **AllSigned**
 - Scripts can run.
 - Requires that all scripts and configuration files be signed by a trusted publisher, including scripts that you write on the local computer.
 - Prompts you before running scripts from publishers that you haven't yet classified as trusted or untrusted.
 - Risks running signed, but malicious, scripts.
- **Bypass**
 - Nothing is blocked and there are no warnings or prompts.
 - This execution policy is designed for configurations in which a PowerShell script is built in to a larger application or for configurations in which PowerShell is the foundation for a program that has its own security model.
- **Default**
 - Sets the default execution policy.
 - Restricted for Windows clients.
 - RemoteSigned for Windows servers.
- **RemoteSigned**
 - The default execution policy for Windows server computers.
 - Scripts can run.
 - Requires a digital signature from a trusted publisher on scripts and configuration files that are downloaded from the internet which includes email and instant messaging programs.
 - Doesn't require digital signatures on scripts that are written on the local computer and not downloaded from the internet.
 - Runs scripts that are downloaded from the internet and not signed, if the scripts are unblocked, such as by using the Unblock-File cmdlet.
 - Risks running unsigned scripts from sources other than the internet and signed scripts that could be malicious.

- **Execution Policy:** Controls the conditions under which PowerShell loads configuration files and runs scripts.
- By default, Windows blocks scripts from running until the policy is changed.

To change the execution policy:

```
Set-ExecutionPolicy -Scope CurrentUser -ExecutionPolicy RemoteSigned
```

Notes:

- Execution policy only applies to PowerShell on Windows.
- Non-Windows platforms do not enforce execution policies.

Execution Policy Documentation

- **Restricted**
 - The default execution policy for Windows client computers.
 - Permits individual commands, but does not allow scripts.
 - Prevents running of all script files, including formatting and configuration files (.ps1xml), module script files (.psm1), and PowerShell profiles (.ps1).
- **Undefined**
 - There is no execution policy set in the current scope.
 - If the execution policy in all scopes is Undefined, the effective execution policy is Restricted for Windows clients and RemoteSigned for Windows Server.
- **Unrestricted**
 - The default execution policy for non-Windows computers and cannot be changed.
 - Unsigned scripts can run. There is a risk of running malicious scripts.
 - Warns the user before running scripts and configuration files that are not from the local intranet zone.

- For a complete description of execution policies and their differences, refer to the help page:

```
Get-Help about_Execution_Policies
```

PowerShell Functions

PowerShell Functions

- A **function** in PowerShell is a grouping of code with optional input and output.
- Functions allow for reusing code by calling it instead of duplicating it multiple times.
- Functions can be thought of as **methods** in object-oriented programming.
- Any part of your script can call a defined function.
- Functions declared will NOT remain if the terminal exits and reopens
 - Functions you want to remain persistent should be defined in the PowerShell **\$PROFILE**

```
Function Write-Something {  
  
    Param($item)  
  
    Write-Host "You passed the parameter $item into the function"  
  
}
```

Creating a PowerShell Function

- A function consists of a list of PowerShell statements with an assigned name.
- To run a function, type the function's name.
- The statements in the function execute as if they were typed directly into the command prompt.

Example:

```
function Get-Hello {  
    Write-Output "Hello, World!"  
}
```

- To call the function:

```
Get-Hello  
# to call a function multiple times use a loop  
for(i; i -le 2;i++){Get-Hello}
```



Something is wrong with this loop. Fix it later.

Advanced Function Example

- Functions can be used to simplify repetitive tasks by consolidating reused code.
- Example: Creating a function to retrieve the DHCP service.


```
Write-Something "Hello!"
```

```
function Get-MultiplicativeResult {  
  
    Param ([int]$a,[int]$b)  
  
    $c = $a * $b  
  
    Write-Output $c  
  
}
```

```
Get-MultiplicativeResult 5 10
```

```
function Get-DHCPService { Get-Service -Name DHCP }
```

Example:

```
function Get-DHCPService {  
    Get-Service -Name DHCP  
}
```

- Call the function like a cmdlet:

```
Get-DHCPService
```

We can then call this function `Get-DHCPService` like a cmdlet.

- **Why use functions?**
 - They streamline code.
 - They make scripts easier to maintain and reuse.

Apply Your Learning

- Open PowerShell and build a custom function:
 - Define the function.
 - Use it to simplify a repetitive task.
 - Test your function by calling it.

```
function Get-FileLoc {
    Param([string]$filter)
    gci -recurse -filter "$filter*"

}

Get-FileLoc host
```

```
function Stop-PowerShell {
    $procs=(Get-WmiObject Win32_Process | Where {$_.Name -eq 'powershell.exe'} `
        | ?{$_.ProcessId -ne $pid} | Select-Object -ExpandProperty ProcessId)

    Do {
        ForEach ($i in $procs) {
            Get-Process powershell | ?{$_.Id -ne $pid} | Stop-Process | Out-Null
            $procs=(Get-WmiObject Win32_Process | Where {$_.Name -eq 'powershell.exe'} `
                | ?{$_.ProcessId -ne $pid} | Select-Object -ExpandProperty ProcessId)
        }
    } While ($procs)
}
```

String Manipulation in PowerShell

- Strings in PowerShell are always **objects**, whether they are literal strings or variables.
- String objects have methods that provide powerful manipulation capabilities.
- Use the `Get-Member` cmdlet to explore available methods:

```
"this is a string" | Get-Member
#also
$string = "a string"
```

```
$string | Get-Member
#you may also call a method on a variable like...
Write-Host $string.ToUpper()
#or
Write-Host $string.Length
```

String Manipulation

- A **variable** is a name that represents an object.
- String literals and string variables both represent **string objects**.
- Use built-in **string methods** to manipulate and interact with string objects.

Select-Object

- The `Select-Object -Property` cmdlet creates a **collection** of user object properties.
- To convert the results into a string for manipulation, further processing is required.

Example:

Concatenation

```
$string = "This is a "
$string + "string"
Write-Output "$string string"
```

```
#The command below selects the collection of Names of users
$users = Get-LocalUser | Select-Object -Property Name

#The command below selects the strings within the collection of users.
$users = Get-LocalUser | Select-Object -ExpandProperty Name
```

```
Get-ADUser -Filter * | Select-Object -Property Name
```

ExpandProperty

- The `Select-Object -ExpandProperty` cmdlet directly converts the property argument (e.g., `Name`) into a string, ready for manipulation.

Example:

```
#The command below selects the strings within the collection of users.
$users = Get-LocalUser | Select-Object -ExpandProperty Name

#The script block of this foreach loop is where concatenation happens.
foreach ($user in $users){
    'User Name : ' + $user
}
```

```
Get-ADUser -Filter * | Select-Object -ExpandProperty Name
```

Replace Operator

- The **replace operator** replaces instances of a string (old values) with a specified string (new values).

```
"Old String" -replace "Old","New"
```

```
$sourstring = "Powershell is dumb and I hate it"
```

```
#The replace operator can be stacked multiple times to replace many items
$sourstring -replace "dumb","awesome" -replace "hate","love"
```

Syntax:

```
$string.Replace("oldValue", "newValue")  
# To view the available methods to use on a string:  
"abc" | Get-Member | Where-Object MemberType -eq Method
```

Example:

```
"This is old text".Replace("old", "new")
```

Join Operator

- The **join operator** concatenates a set of strings into a single string.
- Strings are appended in the order they appear in the command.

```
-join "a", "b", "c"  
  
#=====#  
  
-join ("a", "b", "c")  
  
#=====#  
  
$z = "a", "b", "c"  
-join $z  
  
#=====#  
  
"Windows", "PowerShell", "2.0" -join " "
```

Syntax:

```
-join "string1", "string2", "string3"
```

Example:

```
-join ("Power", "Shell")
```

Split Method

- The **split()** method splits a string into an array based on a specified delimiter (non-regex character).

```
"This.is.an.example.of.string.to.split.." -split "\."
```

Syntax:

```
$string.Split("delimiter")
```

Example:

```
"This,is,a,test".Split(",")
```

Regular Expressions (Regex)

- **Regex** is used across many programming languages to filter strings in large data outputs.
- Regex is a powerful tool, often likened to **[CTRL]+F** on adrenaline.
- Regex can perform tasks ranging from the mundane to the highly complex.

```
#Declare our string to apply our regex filter to.
```

```
$string = "HelloMyNameIS Nick123_55-4-11111-asdf_//123-45-6789-hELLOmYnAME Nick"
```

Syntax	Example	Purpose
<code>+</code>	<code>[a-z]+</code>	Match previous character or character set at least once
<code>?</code>	<code>[c-e]?</code>	Match at most once; previous character or character set may or may not exist
<code>*</code>	<code>App*</code> and <code>Bana*</code>	Any character; match any number of times
<code>{min,max}</code>	<code>[0-9]{1,3}</code>	Match at least <code>min</code> times, and at most <code>max</code> times
<code>{n}</code>	<code>[0-9]{3}</code>	Match exactly <code>n</code> times

Match and Pattern

- In Regex:
 - `\d` matches any number (0-9).
 - `[0-9]` achieves the same as `\d`.
 - `.` is escaped (`\.`) when used as a literal, to avoid being interpreted as a range.
 - `{2,3}` is a **repetition quantifier**, denoting that the preceding pattern must:
 - Appear at least 2 times.
 - Appear up to 3 times.

Example:

#Example 1

```
$string -Match "(\d{2,3}\-){2}\d{2,4}"
```

```
$string -match "\d{2,3}"
```

Repetition Quantifiers

- `{1,}` denotes **one or more** occurrences of a designated pattern.
- The `match` parameter can match a regex pattern against any provided string.

Example:

```
$string -match "\d{1,}"
```

Match and Pattern

- To match a string using the contents of a file(s), use the `Select-String` cmdlet.
- **Pattern VS Match :**
 - `Match` returns a Boolean indication of a match (`$true` or `$false`).
 - `Pattern` (used with `Select-String`) returns the **entire line** containing the matched string.

Example:

```
#Example 1
```

```
$string -Match "(\d{2,3}\-){2}\d{2,4}"
```

```
#Example 2
```

```
$string | Select-String -Pattern "(\d{2,3}\-){2}\d{2,4}"
```

```
#Example 3
```

```
#Foreach loops through all the matches and prints the collection to the screen.
```

```
$string | Select-String -Pattern "(\d{2,3}\-){2}\d{2,4}" | ForEach {$_.Matches.Value}
```



```
Get-Content file.txt | Select-String -Pattern "example"
```

Passing Matches to a ForEach Loop

- Use a `ForEach` loop in conjunction with `.Matches()` and `.Value()` methods to extract only the string that matches the `Pattern` parameter.

Example:

```
Get-Content file.txt | Select-String -Pattern "example" | ForEach-Object { $_.Matches.Value }
```

Regex Example: Pentesting Guide

- Download and store content from a webpage:

```
Invoke-WebRequest -Uri "http://www.pentest-standard.org/index.php/PTES_Technical_Guidelines" -UseBasicParsing |  
Select-Object -ExpandProperty Content |  
Out-File ptes.txt
```

- Check the file size:

```
(Get-Content .\ptes.txt).Length
```

- Use regex to search for the string `"nmap"`:

```
Get-Content .\ptes.txt | Select-String -Pattern "nmap" | ForEach-Object { $_.Matches.Value } | Select -First 2
```

- Extract contextual information:

```
Get-Content .\ptes.txt | Select-String -Pattern "nmap.+" | ForEach-Object { $_.Matches.Value } | Select -First 30 | S
```



```
select -Last 10
```

- Filter for unique matches with a specific pattern:

```
Get-Content .\ptes.txt | Select-String -Pattern "nmap\s\-.  
+" | ForEach-Object { $_.Matches.Value } | Sort-Object -Un  
ique
```

- **Key Takeaway:** Regex allows for efficient filtering and analysis of large files without manual searching. Reinforce to students how powerful regex is for extracting meaningful information.
-

Anchors

- Anchors are regex characters used to match patterns at the **beginning** or **end** of a line.
- **Anchor Characters:**
 - : Searches for patterns at the **beginning** of a line.
 - : Searches for patterns at the **end** of a line.

Examples:


- Match "start" at the beginning of a line:

```
Select-String -Pattern "^start"
```

- Match "end" at the end of a line:

```
Select-String -Pattern "end$"
```

PowerShell Scripts

- A script is a text file containing one or more PowerShell commands, saved with a  extension.

- Scripts allow for automation, reuse, and sharing of commands.

```
function Get-UserInfo {
    param($shadowfile)
    $shadow = get-content $shadowfile

    foreach ($shadowline in $shadow) {

        $username = ($shadowline -split ":")[0]
        $userid = ($shadowline -split ":")[2]
        $groupid = ($shadowline -split ":")[3]
        $fullname = ($shadowline -split ":")[4]
        $userhomedir = ($shadowline -split ":")[5]
        $defaultShell = ($shadowline -split ":")[6]

        Write-Output ("
        User Name: `t $username
        User Id: `t $userid
        GroupId: `t $groupid
        Name: `t $fullname
        Home : `t $userhomedir
        Shell: `t $defaultShell `n")
    }
}
```

```
Get-UserInfo `path-to-shadow-file`
```

```
#Gets all the process on a local machine and stores the output into a file name processes.txt in the current directory.
#===
get-process | Out-File -FilePath .\processes.txt

#open the file to see the contents.
#===
cat .\processes.txt

#Changes the occurrences of the string "svchost" to XXXX and overwrites the file with the changes.
#===
$content = (Get-Content -Path .\processes.txt)
$content -replace "svchost","XXXX" | Set-Content -path .\processes.txt
#===
#open the file to see the changed contents.
cat .\processes.txt
```

Search and Replace Strings in a File

- Use PowerShell to search and replace strings within a file.

Example:

```
(Get-Content file.txt).Replace("oldString", "newString") | Set-Content file.txt
```



Final Challenge Problem

From 1-100

Every # divisible by 3, print #-skibbidi

Every # divisible by 5, print #-toilet

Every # divisible by 3 AND 5, print #-rizzler

Every # not in the above, print #

Solution Algorithms for PE Problems

Count the number of occurrences of a pattern in a file:

```
(Get-Content \Path\to\file | Select-String -Pattern "Regular Expression").Matches.Count
```

- You may wish to substitute `-Pattern` for `-AllMatches`, `-CaseSensitive`, ...



When to use `-AllMatches`: when you may want to capture more than one instance of a pattern on a line. Other options tend to select the first instance.

To troubleshoot your solution and consider edge cases, you can view each item in the count printed to the console using:

```
(Get-Content \Path\to\file | Select-String -Pattern "Regular Expression").Matches | Select-Object -ExpandProperty Value
```

Count the number of Unique occurrences of a pattern in a file:

```
(Get-Content \Path\to\file | Select-String -Pattern "Regular Expression" -AllMatches | Select-Object -ExpandProperty Matches | ForEach-Object { $_.Value } | Sort-Object -Unique).Count
```

1. **Define the pattern**
2. **Search for the pattern in the file:** Using PowerShell's `Select-String` cmdlet with regex.
3. **Filter for unique matches:** Using `Sort-Object -Unique`. You *must* sort before you can look for unique entries
4. **Count the unique matches:** Using `.Count`.

Which of the following is not a [member] available in [cmdlet] -type questions

```
#search_term is an array containing all the possible multiple choice answers
$search_term = @("Possible", "answers")
#search_block is an array containing all of the possible members, created via pipelining. The command can be substituted for any other, as can the property filtered for.
$search_block = @(Get-Help | Get-Member | Where-Object -Property MemberType -eq Method)
```

```
#search_block names extracts the name data specifically from
search_block.
$search_block_names = $search_block.Name

#this loop iterates over each term in search_term and checks
for it in the search_block_names, then identifies the missing
term.
foreach ($term in $search_term){
    if (-not($search_block_names -contains $term)) {
        Write-Output "Not found: $term"
    }
}
```

Composable Resources

Regex Patterns

1. Email Addresses

Pattern: An email address typically consists of a username, an @ symbol, and a domain.

Regex:

```
\b[A-Za-z0-9._%+-]+\@[A-Za-z0-9.-]+\.[A-Za-z]{2,}\b
```

Explanation:

- `\b` : Word boundary.
- `[A-Za-z0-9._%+-]+` : Matches the username (letters, numbers, and special characters).
- `@` : Matches the @ symbol.
- `[A-Za-z0-9.-]+` : Matches the domain name.
- `\.[A-Za-z]{2,}` : Matches the top-level domain (e.g., `.com`, `.org`).

2. Dates (MM/DD/YYYY or DD-MM-YYYY)

Pattern: Matches common date formats like `12/31/2023` or `31-12-2023`.

Regex:

```
\b(0[1-9]|1[0-2])[-\/](0[1-9]|[12][0-9]|3[01])[-\/](\d{4})\b
```

Explanation:

- `(0[1-9]|1[0-2])` : Matches months (`01` to `12`).
- `[-\/]` : Matches either a dash (`-`) or a slash (`/`) as a separator.
- `(0[1-9]|[12][0-9]|3[01])` : Matches days (`01` to `31`).
- `(\d{4})` : Matches a four-digit year.

3. Times (HH:MM, 24-Hour Format)

Pattern: Matches time in the 24-hour format like `23:59`.

Regex:

```
\b([01][0-9]|2[0-3]):[0-5][0-9]\b
```

Explanation:

- `([01][0-9]|2[0-3])` : Matches hours (`00` to `23`).
- `: [0-5][0-9]` : Matches minutes (`00` to `59`).

4. URLs

Pattern: Matches valid URLs, including `http`, `https`, and `www`.

Regex:

```
\b(https?:\/\/)?(www\.)?[A-Za-z0-9.-]+\.[A-Za-z]{2,}(\S*)\b
```

Explanation:

- `(https?:\/\/)?` : Matches optional `http://` or `https://`.

- `(www\.)?` : Matches optional `www.`.
- `[A-Za-z0-9.-]+` : Matches the domain name.
- `\.[A-Za-z]{2,}` : Matches the top-level domain.
- `(\S*)` : Matches the rest of the URL.

5. Social Security Numbers (SSNs)

Pattern: Matches US SSNs in the format `123-45-6789`.

Regex:

```
\b\d{3}-\d{2}-\d{4}\b
```

Explanation:

- `\d{3}` : Matches the first three digits.
- `-` : Matches the first dash.
- `\d{2}` : Matches the middle two digits.
- `-` : Matches the second dash.
- `\d{4}` : Matches the last four digits.

6. MAC Addresses

Pattern: Matches MAC addresses like `00:1A:2B:3C:4D:5E`.

Regex:

```
\b([A-Fa-f0-9]{2}[:-]){5}[A-Fa-f0-9]{2}\b
```

Explanation:

- `([A-Fa-f0-9]{2}[:-]){5}` : Matches the first five groups of two hexadecimal characters separated by `:` or `-`.
- `[A-Fa-f0-9]{2}` : Matches the last two hexadecimal characters.

7. Credit Card Numbers

Pattern: Matches 16-digit credit card numbers with optional spaces or dashes.

Regex:

```
\b\d{4}[-\s]?d{4}[-\s]?d{4}[-\s]?d{4}\b
```

Explanation:

- `\d{4}` : Matches a group of four digits.
- `[-\s]?` : Matches an optional dash () or space.
- Repeated 4 times for 16 digits.

8. US ZIP Codes

Pattern: Matches ZIP codes in the formats `12345` or `12345-6789`.

Regex:

```
\b\d{5}(-\d{4})?\b
```

Explanation:

- `\d{5}` : Matches the first five digits.
- `(-\d{4})?` : Matches an optional hyphen followed by four digits.

9. Hexadecimal Numbers

Pattern: Matches hexadecimal numbers, e.g., `0x1A3F`.

Regex:

```
\b0x[A-Fa-f0-9]+\b
```

Explanation:

- `0x` : Matches the prefix for hexadecimal numbers.
- `[A-Fa-f0-9]+` : Matches one or more hexadecimal digits.

10. Floating-Point Numbers

Pattern: Matches numbers with optional decimals, e.g., `123.45` or `0.5`.

Regex:

```
\b\d+(\.\d+)?\b
```

Explanation:

- `\d+` : Matches one or more digits.
- `(\.\d+)?` : Matches an optional decimal point followed by digits.

11. RegEx for an Area Code Optionally Enclosed in Parentheses

Pattern: Matches a three-digit area code, optionally enclosed in parentheses.

Regex:

```
\(?:\d{3}\)?
```

Instructions:

- `\(?:` : Matches an optional opening parenthesis (().
- `\d{3}` : Matches exactly three digits.
- `\)?` : Matches an optional closing parenthesis ()).

Example Matches:

- `(123)`
- `456`
- `(987)`

12. RegEx for a Telephone Number

Pattern: Matches a standard telephone number, with flexibility for separators (spaces or dashes) and optional parentheses around the area code.

Regex:

```
\(?:\d{3}\)?[-\s]?\d{3}[-\s]?\d{4}
```

Instructions:

- `\(?:\d{3}\)?` : Matches a three-digit area code, optionally enclosed in parentheses.
- `[-\s]?` : Matches an optional separator (dash or space).
- `\d{3}` : Matches the next three digits.
- `[-\s]?` : Matches another optional separator.
- `\d{4}` : Matches the final four digits of the phone number.

Example Matches:

- `(123) 456-7890`
- `123-456-7890`
- `123 456 7890`
- `(987)-654-3210`

13. RegEx for an IP Address

Pattern: Matches a valid IPv4 address (four groups of numbers between 0 and 255, separated by dots).

Regex:

```
\b((25[0-5]|2[0-4][0-9]|1[0-9]{2}|[1-9][0-9]|[0-9])\.){3}(25[0-5]|2[0-4][0-9]|1[0-9]{2}|[1-9][0-9]|[0-9])\b
```

Instructions:

- `\b` : Ensures the match starts and ends on a word boundary.
- `(25[0-5]|2[0-4][0-9]|1[0-9]{2}|[1-9][0-9]|[0-9])` :
 - Matches a number from 0 to 255.
 - `25[0-5]` : Matches 250–255.

- `2[0-4][0-9]` : Matches 200–249.
- `1[0-9]{2}` : Matches 100–199.
- `[1-9][0-9]` : Matches 10–99.
- `[0-9]` : Matches 0–9.
- `\.` : Matches the literal dot separating the octets.
- `{3}` : Repeats the pattern for the first three octets.
- `\b` : Ensures the match ends cleanly.

Example Matches:

- `192.168.0.1`
- `10.0.0.1`
- `255.255.255.255`
- `127.0.0.1`

RegEx Cheat Sheet:

[regular-expressions.pdf](#)

Show Data-type

```
(Get-Process ssh | Select-Object -Property Handles).GetType()
# returns PSCustomObject
# To better understand the output of a pipeline Use -ExpandProperty
(Get-Process ssh | Select-Object -ExpandProperty Handles).GetType()
#Also use -ExpandProperty to get more info about an object.
```

PowerShell Cmdlets

How to open a searchable window with Get-Help?

Get-Help about_Regular_Expressions -ShowWindow

How to update your Get-Help Cmdlet?

Update-Help -Verbose -Force -ErrorAction SilentlyContinue

What does the get-member cmdlet do?

The Get-Member cmdlet gets the members, the properties and methods, of objects.

True or False: Parameter sets are not mutually exclusive.

False (They are mutually exclusive!)

How to check the PowerShell version?

Run the '\$PSVersionTable' command

How to display usage examples of a specific command?

Get-Help <Your Command> -Examples

Get-Help -Examples Get-Location or Get-Help Get-Location -Examples

How to get an alias for a command?

Get-Alias -Definition Get-ChildItem

To show all cmdlets available in PowerShell:

Get-Command

How to get the command of an alias?

Get-Command cls

What is the command to count the empty lines in the HR file?

(\$employees | Where-Object {\$_ -match “^\s*\$”}).count

\$employees | Where-Object {\$_ -match “^\s*\$”} | Measure-Object

List all available commands related to “-Object”.

Get-Command -Noun Object

List all available commands related to "Get-":

Get-Command -Verb Get

What command will show you all the occurrences of "." in the HR file?

(\$employees | Select-String -Pattern "." -AllMatches).Matches.Count

Define file path

\$filePath = "\$HOME\Desktop\PowerShell_Commands.txt"

To check your execution policy:

Get-ExecutionPolicy

To change execution policy (allow scripts to run):

Set-ExecutionPolicy RemoteSigned -Scope CurrentUser

To navigate to Desktop:

cd \$HOME\Desktop

To run a PowerShell script:

.\script.ps1

To bypass execution policy for one script run:

powershell -ExecutionPolicy Bypass -File script.ps1

To write output to the console:

Write-Output "Hello, PowerShell!"

To ask for user input:

\$userName = Read-Host "What is your name?"

To check if a file or folder exists:

Test-Path C:\Users\\$env:UserName\Desktop\MyFile.txt

To create a text file and write content to it:

**"This is a test file" | Out-File -FilePath
C:\Users\\$env:UserName\Desktop\MyFile.txt**

To list all running services:

Get-Service | Where-Object { \$_.Status -eq 'Running' }

To start a specific service:

Start-Service -Name Spooler

To stop a specific service:

Stop-Service -Name Spooler

To count how many times a pattern appears in a file:

(Get-Content C:\Users\%env:UserName\Desktop\MyFile.txt | Select-String -Pattern "word" -AllMatches).Matches.Count

To find all '.com' domains in a file:

(Get-Content C:\Users\%env:UserName\Desktop\MyFile.txt | Select-String -Pattern "\.com\b" -AllMatches).Matches.Value

To search for emails in a file:

(Get-Content C:\Users\%env:UserName\Desktop\MyFile.txt | Select-String -Pattern "\b[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\.[A-Za-z]{2,}\b" -AllMatches).Matches.Value

To check if a registry key exists:

Test-Path HKLM:\Software\Microsoft\Windows\CurrentVersion

Save commands to a text file

\$commands | Out-File -FilePath \$filePath

Define file path

\$filePath = "\$HOME\Desktop\PowerShell_Commands.txt"

To check your execution policy:

Get-ExecutionPolicy

To change execution policy (allow scripts to run):

Set-ExecutionPolicy RemoteSigned -Scope CurrentUser

To bypass execution policy for a single script run:

powershell -ExecutionPolicy Bypass -File script.ps1

To navigate to the Desktop:

cd \$HOME\Desktop

To check if a file or folder exists:

Test-Path C:\Users\%env:UserName\Desktop\MyFile.txt

To create a text file and write content to it:

**"This is a test file" | Out-File -FilePath
C:\Users\%env:UserName\Desktop\MyFile.txt**

To delete a file:

Remove-Item C:\Users\%env:UserName\Desktop\MyFile.txt

To convert a string to uppercase:

**\$raj = "I am a programmer."
\$raj.ToUpper()**

To count occurrences of "." in a file:

**(Get-Content C:\Users\%env:UserName\Desktop\MyFile.txt -Raw) -split '\.' |
Measure-Object | Select-Object -ExpandProperty Count**

To count how many times a pattern appears in a file:

**(Get-Content C:\Users\%env:UserName\Desktop\MyFile.txt | Select-String -Pattern
"word" -AllMatches).Matches.Count**

To find all ".com" domains in a file:

**(Get-Content C:\Users\%env:UserName\Desktop\MyFile.txt | Select-String -Pattern
"\.com\b" -AllMatches).Matches.Value**

To search for emails in a file:

**(Get-Content C:\Users\%env:UserName\Desktop\MyFile.txt | Select-String -Pattern
"\b[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\.[A-Za-z]{2,}\b" -AllMatches).Matches.Value**

To check if a line starts with "error":

\$message -match "^error"

To list all running services:

Get-Service | Where-Object { \$_.Status -eq 'Running' }

To list all running services and filter further

Get-Service | Select-Object Name, Status, DisplayName

To get all properties and methods of an object (example: services):

Get-Service | Get-Member

To show all cmdlets available in PowerShell:

Get-Command

To ask for user input and display a greeting:

\$userName = Read-Host "What is your name?"

Write-Output "Hello, \$userName!"

To escape double quotes inside a string:

Write-Output "He said, `"PowerShell is great!`""

To insert a newline in a string:

Write-Output "Line1`nLine2"

To insert a tab in a string:

Write-Output "Column1`tColumn2"

To list all environment variables:

Get-ChildItem Env:

To display a specific environment variable (example: username):

\$env:UserName

To install WSL (Windows Subsystem for Linux):

wsl --install

To list available WSL distributions:

wsl --list --online

To install a specific WSL distribution (example: Ubuntu 22.04):

wsl --install -d Ubuntu-22.04

To copy a file:

Copy-Item C:\Users\\$env:UserName\Desktop\MyFile.txt

C:\Users\\$env:UserName\Documents

To move a file:

**Move-Item C:\Users\%env:UserName\Desktop\MyFile.txt
C:\Users\%env:UserName\Documents**

To rename a file:

**Rename-Item C:\Users\%env:UserName\Desktop\MyFile.txt -NewName
"NewFile.txt"**

To check if a registry key exists:

Test-Path HKLM:\Software\Microsoft\Windows\CurrentVersion

To get the current date and time:

Get-Date

To check if PowerShell is running as administrator:

[Security.Principal.WindowsPrincipal]

**[Security.Principal.WindowsIdentity]::GetCurrent().IsInRole([Security.Principal.Wi
ndowsBuiltInRole]::Administrator)**

Functions

#1

```
function Find-ReplaceText {  
    param (  
        [string]$filePath,    # File to modify  
        [string]$searchWord,  # Word to find  
        [string]$replaceWord  # Word to replace with  
    )  
  
    # Check if file exists  
    if (!(Test-Path -Path $filePath)) {  
        Write-Output "❌ Error: File not found at '$filePath'"  
        return  
    }  
  
    # Read file content  
    $content = Get-Content -Path $filePath -Raw  
  
    # Replace the word  
    $updatedContent = $content -replace $searchWord, $replaceWord  
  
    # Save the modified content back to the file  
    $updatedContent | Set-Content -Path $filePath  
  
    Write-Output "✅ Successfully replaced '$searchWord' with '$replaceWord' in  
'$filePath'. "  
}
```

Run the script to load the function: .\find-replace.ps1

To call the function: Find-ReplaceText -filePath

**"C:\Users\YourUser\Desktop\MyFile.txt" -searchWord "oldword" -replaceWord
"newword"**

#2

```
function Get-Hello {  
    Write-Output "Hello, World!"  
}
```

```
#this is a for-loop  
for ($i = 1; $i -le 5; $i++) {  
    Get-Hello  
}
```

#3

```
function Get-Hello {  
    Write-Output "Hello, World!"  
}  
#Using while-loop  
$count = 0  
while ($count -lt 5) {  
    Get-Hello  
    $count++  
}
```

#4

```
function Count-WordOccurrences {  
    param (  
        [string]$filePath,    # Path to the file  
        [string]$word        # Word to count  
    )  
  
    # Check if file exists  
    if (!(Test-Path -Path $filePath)) {  
        Write-Output "❌ Error: File not found at '$filePath'"  
        return  
    }  
  
    # Read file content  
    $content = Get-Content -Path $filePath -Raw  
  
    # Use regex to find matches (case-insensitive)  
    $matches = [regex]::Matches($content, "\b$word\b", "IgnoreCase")  
  
    # Output the count  
    Write-Output "✅ The word '$word' appears $($matches.Count) times in '$filePath'."  
}
```

Run the script to load the function: .\count-word.ps1

To call the function with parameters: Count-WordOccurrences -filePath "C:\Users\YourUser\Desktop\MyFile.txt" -word "PowerShell"

- PEs are a reflection of what the test looks like

Tips n' tricks

- Will have to make an alias of an alias (like an alias of the man alias (newMan -> man -> help))
- **CTRL+Spacebar (shows you a dropdown of your possible entries based on what you already have typed)**
- Get-help <command> -ShowWindow: brings out a searchable window to find the info you want easier

REGEX

- (Get-Content .\file | Select-String -CaseSensitive -Pattern \$regex -AllMatches).Matches.Value.Count
 - Regex in count
- (Get-Content \$file | Select-String -CaseSensitive -Pattern \$regex -AllMatches | ForEach-Object {\$_.Matches.Value})
 - Regex match
- ((Get-Content .\file | Select-String -CaseSensitive -Pattern \$regex -AllMatches).Matches.Value | Sort-Object -Unique).Count
 - Regex unique
- What is Powershell?
 - PowerShell is a cross-platform automation solution made up of a command-line shell, a scripting language, and a configuration management framework. **PowerShell runs on Windows, Linux, and macOS.**
 - As a scripting language, PowerShell is commonly used for automating the management of systems. It is also used to build, test, and deploy solutions, often in CI/CD environments.
 - In software engineering, CI/CD or CIGD is the combined practice of continuous integration and either continuous delivery or continuous deployment. CI/CD bridges the gaps between development and operation activities and teams by enforcing automation in building, testing and deployment of applications.
 - **PowerShell is built on the .NET Common Language Runtime (CLR).**
 - **All inputs and outputs are .NET objects.**
 - **An object in Powershell is simply an enriched data container.**
 - This container represents data in an organized structure with properties and functions / methods.
 - The methods are actions that can be taken by an object.
 - Core cmdlets in PS are built in .NET Core
- Powershell Features

- PowerShell shares some features with traditional shells:
 - **Built-in help system:**
 - The help system in PowerShell provides information about commands and also integrates with online help articles.
 - **Pipeline:**
 - MA pipeline is used to run many commands sequentially.
 - **Aliases:**
 - Aliases are alternate names that can be used to run commands.
- **How is PS Different?**
 - It operates on objects over text. In a command-line shell, you have to run scripts whose output and input might differ. So you end up spending time formatting the output and extracting the data you need. By contrast, in PowerShell you use objects as input and output. That means you spend less time formatting and extracting. Objects retain their properties even as a user manipulates the output they see on their screen or use in pipelines.
 - **Commands in PowerShell are called cmdlets** (pronounced commandlets). Unlike many other shell environments,
 - **In PowerShell, cmdlets are built on a common runtime rather than separate executables.**
 - Cmdlets typically take object input and return objects.
 - The core cmdlets in PowerShell are built in .NET Core and are open source.
 - You can build your own cmdlets in .NET Core or PowerShell.
 - It has many types of commands. Commands in PowerShell can be native executables, cmdlets, functions, scripts, or aliases.
 - Every command you run belongs to one of these types.
 - The words command and cmdlet are often used interchangeably.

COL

What are some features PowerShell shares with traditional shells?

Built in help system

Pipeline

aliases

What are commands in PowerShell called?

Cmdlets (commandlets)

- **Commandlets**
 - A cmdlet is a lightweight command that is used in the PowerShell environment.
 - The PowerShell runtime invokes these cmdlets within the context of automation scripts that are provided at the command line.

- Cmdlets differ from commands in other command-shell environments in the following ways:
 - **Cmdlets are instances of .NET classes; they are not stand-alone executables.**
 - **Cmdlets process input objects from the pipeline rather than from streams of text, and cmdlets typically deliver objects as output to the pipeline.**
- **Cmdlets employ a verb/noun naming convention** that is designed to make each cmdlet easier to remember and read.
 - As an example, a typical Get-ChildItem command uses the verb Get followed by the noun ChildItem.
- When executed through the PowerShell runtime environment, the Get-ChildItem command lists or returns the items in one or more specified locations.
 - If items are in a container, the command gets the items inside the container — child items.

● PS Pipeline

- Simply put, a pipe | takes the output of the first part of the pipeline, and then uses it as the input of the next part.

```
Steve: C:\Users\student >>>get-service | Select-Object Name
Name
----
AarSvc_4639e
AJRouter
ALG
```

- Using the Select-Object cmdlet, how would we display only the DisplayName Property?

● Parameters and Arguments

- Most cmdlets support the use of parameters as part of the input mechanism. Parameters can be added to the cmdlet at the command line or passed to cmdlets through the pipeline as the output from a previous cmdlet.
- The arguments or values of each parameter detail the actual input that the cmdlet will accept, how the cmdlet should work and what, if any data the cmdlet outputs.
- On the command line, when you run the Get-Service cmdlet, you get a list of services on your machine.
- You can further filter these just to show the services that are running:
 - **Get-Service | Where-Object {\$_.Status -eq "Running"}**


```
PS C:\Users\cvte1> Get-Service | Where-Object {$_.Status -eq "Running"}

Status Name DisplayName
-----
Running Appinfo Application Information
Running AudioEndpointBu... Windows Audio Endpoint Builder
Running Audiosrv Windows Audio
Running BFE Base Filtering Engine
Running BrokerInfrastru... Background Tasks Infrastructure Ser...
Running camsvc Capability Access Manager Service
Running cbdhsvc_4154e Clipboard User Service_4154e
Running CDPSvc Connected Devices Platform Service
Running CDPUserSvc_4154e Connected Devices Platform User Ser...
Running CertPropSvc Certificate Propagation
Running CoreMessagingRe... CoreMessaging
Running CryptSvc Cryptographic Services
Running DcomLaunch DCOM Server Process Launcher
Running Dhcp DHCP Client
Running DiagTrack Connected User Experiences and Tele...
Running DispBrokerDeskt... Display Policy Service
Running Dnscache DNS Client
Running DPS Diagnostic Policy Service
Running DsSvc Data Sharing Service
Running DsmSvc Data Usage
Running EventLog Windows Event Log
Running EventSystem COM+ Event System
Running FontCache Windows Font Cache Service
Running InstallService Microsoft Store Install Service
Running iphlpsvc IP Helper
Running KeyIso CNG Key Isolation
Running LanmanServer Server
Running LanmanWorkstation Workstation
Running Tfsvc Geolocation Service
Running LicenseManager Windows License Manager Service
Running lmhosts TCP/IP NetBIOS Helper
Running LSM Local Session Manager
Running MDCoreSvc Microsoft Defender Core Service
Running mpssvc Windows Defender Firewall
Running MSDTC Distributed Transaction Coordinator
Running NcbService Network Connection Broker
Running netprofm Network List Service
Running NPSMSvc_4154e NPSMSvc_4154e
```

COL

What type of naming convention do cmdlets use?

verb-noun

Which statement is true about how cmdlets differ from commands?

- a. Cmdlets are stand-alone executables.
- b. Cmdlets are instances of .NET classes**
- c. both a & b

- **PS Help System**

- Most shells have some kind of help system in which you can learn more about a command. For example, you can learn what the command does and what parameters it supports.

- **Powershell Help Cmdlets:**

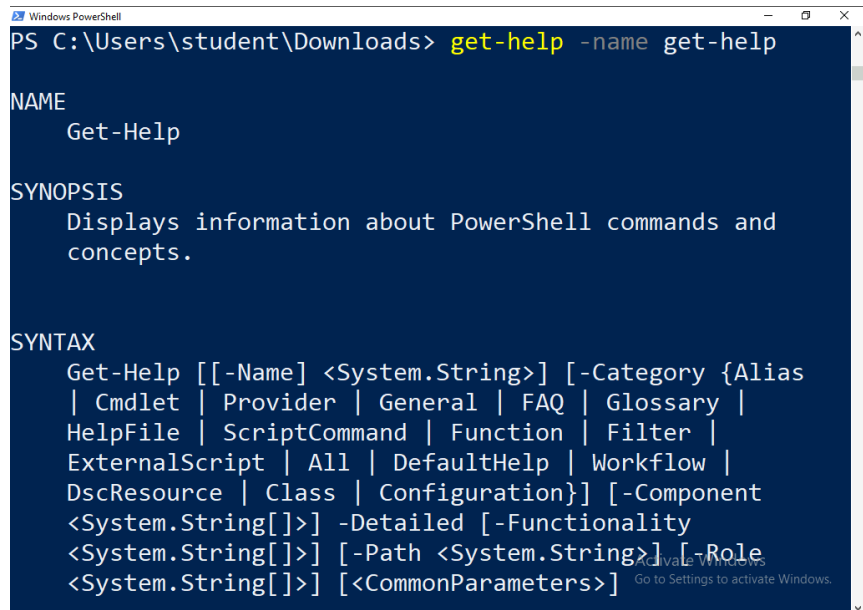
- **Get-Help**

- Get-Help is a multipurpose command. Get-Help helps you learn how to use commands once you find them.
- When Get-Help is used to locate commands, it first searches for wildcard matches of command names based on the provided

input. If it doesn't find a match, it searches through the help topics themselves, and if no match is found an error is returned.

- **Get-Help -Name Get-Help**

- We are calling "Get-Help" on the cmdlet named "Get-Help"



```
PS C:\Users\student\Downloads> get-help -name get-help

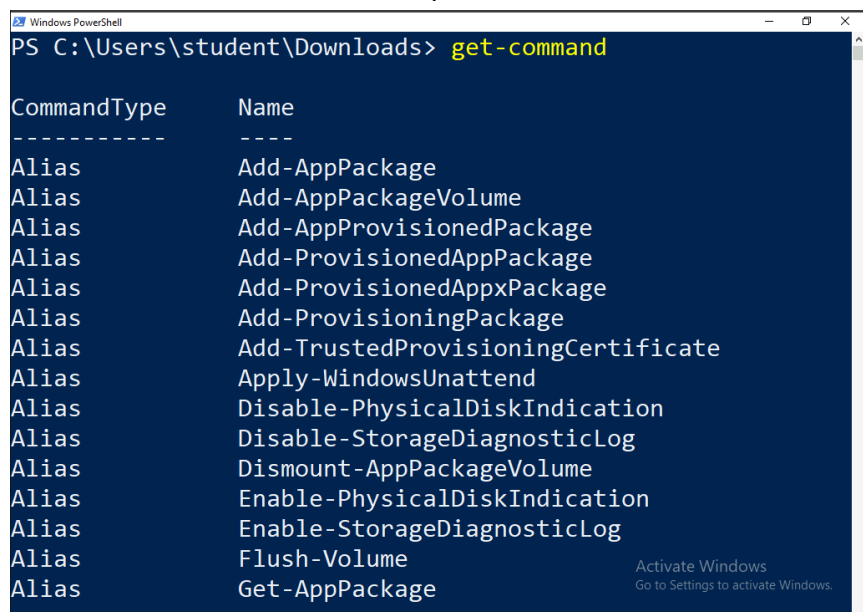
NAME
    Get-Help

SYNOPSIS
    Displays information about PowerShell commands and concepts.

SYNTAX
    Get-Help [[-Name] <System.String>] [-Category {Alias | Cmdlet | Provider | General | FAQ | Glossary | HelpFile | ScriptCommand | Function | Filter | ExternalScript | All | DefaultHelp | Workflow | DscResource | Class | Configuration}] [-Component <System.String[]>] [-Detailed [-Functionality <System.String[]>] [-Path <System.String>] [-Role <System.String[]>] [-CommonParameters>]
```

- **Get-Command**

- The Get-Command cmdlet gets all commands that are installed on the computer, including cmdlets, aliases, functions, filters, scripts, and applications.
- This is helpful when we want to get a list of all the commands that are available to us in the current powershell session.

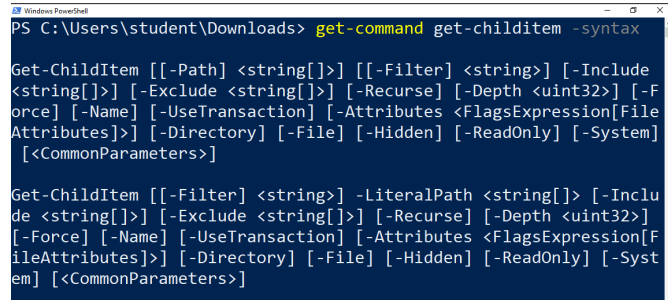


```
PS C:\Users\student\Downloads> get-command

CommandType      Name
-----
Alias             Add-AppPackage
Alias             Add-AppPackageVolume
Alias             Add-AppProvisionedPackage
Alias             Add-ProvisionedAppPackage
Alias             Add-ProvisionedAppxPackage
Alias             Add-ProvisioningPackage
Alias             Add-TrustedProvisioningCertificate
Alias             Apply-WindowsUnattend
Alias             Disable-PhysicalDiskIndication
Alias             Disable-StorageDiagnosticLog
Alias             Dismount-AppPackageVolume
Alias             Enable-PhysicalDiskIndication
Alias             Enable-StorageDiagnosticLog
Alias             Flush-Volume
Alias             Get-AppPackage
```

- **Get-childterm**

- This command grabs the parameter sets of the Get-ChildItem command
- **Syntax: Get-Command -Name Get-Childitem -Syntax**
- We can replace Get-ChildItem with any command we want to get the syntax for.
- We are calling "Get-command" on the cmdlet named "Get-Childitem" to see what the accepted syntax is.



```

PS C:\Users\student\Downloads> get-command get-childitem -syntax

Get-ChildItem [[-Path] <string[]>] [[-Filter] <string>] [-Include <string[]>] [-Exclude <string[]>] [-Recurse] [-Depth <uint32>] [-Force] [-Name] [-UseTransaction] [-Attributes <FlagsExpression[FileAttributes]>] [-Directory] [-File] [-Hidden] [-ReadOnly] [-System] [<CommonParameters>]

Get-ChildItem [[-Filter] <string>] -LiteralPath <string[]> [-Include <string[]>] [-Exclude <string[]>] [-Recurse] [-Depth <uint32>] [-Force] [-Name] [-UseTransaction] [-Attributes <FlagsExpression[FileAttributes]>] [-Directory] [-File] [-Hidden] [-ReadOnly] [-System] [<CommonParameters>]

```

■ Get-Member

- The Get-Member cmdlet gets the members, the properties and methods, of objects.
- **Syntax: get-process | get-member**
- Takes Get-Process and pipes it into Get-Member so that we can see all the methods and properties of that object. Think of the methods as “things we can do with these objects,” and properties as “things these objects are.”

● PS Objects

- Made up of three types of data:
 - Object type
 - Properties
 - Methods

● Parameters:

- Parameters:
 - NAME
 - SYNOPSIS
 - SYNTAX
 - DESCRIPTION
 - RELATED LINKS
 - REMARKS
- As you can see, help topics can contain an enormous amount of information and this isn't even the entire help topic.

- Take a moment to run that example on your computer, review the output, and take note of how the information is grouped:
- While not specific to PowerShell, a parameter is a way to provide input to a command. Get-Help has many parameters that can be specified in order to return the entire help topic or a subset of it.
- The syntax section of the help topic shown in the previous set of results lists all of the parameters for Get-Help.
- At first glance, it appears the same parameters are listed six different times. Each of those different blocks in the syntax section is a parameter set. This means the Get-Help cmdlet has six different parameter sets. If you take a closer look, you'll notice that at least one parameter is different in each of the parameter sets.

- **Parameter Sets**

- Parameter sets are mutually exclusive. Once a unique parameter that only exists in one of the parameter sets is used, only parameters contained within that parameter set can be used. For example, both the Full and Detailed parameters couldn't be specified at the same time because they are in different parameter sets.
- Each of the following parameters are in different parameter sets:
 - Full
 - Detailed
 - Examples
 - Online
 - Parameter
 - ShowWindow

- **Wildcard**

- A wildcard character "*", simply matches any and all string filters.
- **Syntax: Get-Command Get***
- We are calling "Get-command" and filtering on command that starts with Get

COL

What does the get-member cmdlet do?

Gets members (properties, methods, etc) of an object

True or False: Parameter sets are not mutually exclusive.

False

- **PS Aliases**

- An alias is an alternate name for a cmdlet, function, executable file, including scripts. PowerShell includes a set of built-in aliases. You can add your own aliases to the current session and to your PowerShell profile.

- PowerShell supports the use of common aliases such as cls (clear the screen) and ls (list the files). PowerShell includes built-in aliases that are available in each PowerShell session. So new users can use their knowledge of other frameworks and don't necessarily have to remember the PowerShell name for familiar commands.
- We can add new aliases for our own session of Powershell.
- To create an alias, use the cmdlets Set-Alias or New-Alias.
- Syntax: **Set-Alias -Name list -Value Get-Childitem**
- We can verify our alias was created using the Get-Alias cmdlet.
- **Syntax: Get-Alias -Name list**
- We have now created an alias called list. This will function EXACTLY the same as Get-Childitem.
- Using the Definition parameter, you can find all the aliases of a command. Let's find all the current aliases of the Get-Childitem cmdlet.

COL

What is the full command to create an alias?

Set-alias -name <aliasNAME> -value <cmdNAME>

True or False: The get-alias cmdlet can verify an alias was created

True

• Variables

- A variable is a unit of memory in which values are stored. In PowerShell, variables are represented by text strings that begin with a dollar sign (\$), such as \$a, \$process, or \$my_var.
- Variable names are not case-sensitive in Powershell, and can include spaces and special characters.

○ 3 types of Variables

■ User Created Variables

- **User-created variables are created and maintained by the user.** By default, the variables that you create at the PowerShell command line exist only while the PowerShell window is open.
- When the PowerShell window is closed, the variables are deleted.
- To save a variable, add it to your PowerShell profile.
- You can also create variables in scripts with global, script, or local scope.

```
#User-created variable
$my_var = 1
echo $my_var

$my_var += 1
echo $my_var
```

■ Automatic Variables

- **Automatic variables store the state of PowerShell.** These variables are created by PowerShell, and PowerShell changes their values as required to maintain their accuracy.
- **Users can't change the value of these variables.**
 - For example, the \$PSHOME variable stores the path to the PowerShell installation directory.

```
#Automatic variable
echo $PSVersionTable
echo $PSHome
```

■ Preference Variables

- **Preference variables store user preferences for PowerShell.** These variables are created by PowerShell and are populated with default values.
- **Users can change the values of these variables.**
 - For example, the \$MaximumHistoryCount variable determines the maximum number of entries in the session history.

```
#Preference variable
echo $MaximumAliasCount
echo $MaximumHistoryCount
```

COL

List some of the different types of variables discussed in Powershell

User created
Automatic
Preference

True or False: Users can change preference variables
true

- Arrays

- An array is a data structure that is designed to store a collection of items. The items can be the same type or different types. An array is organized by a zero-based index.
- To create and initialize an array, assign multiple values to a variable.
- The values stored in the array are delimited with a comma and separated from the variable name by the assignment operator (=).

```
$A = 1,5,10,15,10,"squirrel"
#In this example we are taking the values 22,5,10,8,12,9 and 80 and storing them into the array named $A
■ echo $A
```

- This data is now associated in memory with the array named \$A. Anytime we want to access this information we can interact with the array it is stored in.

```
#We can access the value at a specific position by referring to its index.
■ echo $A[1]
```

- The value 5 is stored at index 1.
- \$A[-2] would be equal to 10 because it basically starts at the back of the array and goes from there for negative indices

● Array Sub-Expressions

- The array sub-expression operator creates an array from the statements inside it. Whatever the statement inside the operator produces, the operator will place it in an array. Even if there is zero or one object.

```
Sub Expression Syntax
@( commands )

Sub Expression Syntax
$services = @(get-service | where {$_.Status -eq "Running"} | Select-Object Name )
■
```

- Now we have an array named \$services that is a collection of all the service names with a status of "Running."
- We can now interact with our service names as if they were just a regular array.

```
■ echo $services[12]
```

- We've taken output from a commandlet and have organized it into an array for further data processing or analyzing.
- One of the properties of an array is the length. This is simply how many objects are in an array. We can access this value using the .length method of an array.

● Hash Tables

- A hash table, also known as a dictionary or associative array, is a compact data structure that stores one or more key/value pairs.
- For example, a hash table might contain a series of IP addresses and computer names, where the IP addresses are the keys and the computer names are the values, or vice versa.

- To create a hash table, follow this syntax:

```
■ $phonebook = @{ Bob = "706-123-4567"; Alice = "803-123-4567"; Steve = "555-123-4567" }
```

- We now have an array where keys are associated with their corresponding values.
- Unlike a regular array hash tables use key based indexing. We cannot refer to values in their numerical position. We have to reference the specific key in order to access the value.

```
■ echo $phonebook["Bob"]  
echo $phonebook["Alice"]
```

COL

What is an array?

data structure that is designed to store a collection of items.

What are the values stored in an array delimited with?

A comma

● Powershell Features

- You can create a PowerShell profile to customize your environment and to add session-specific elements to every PowerShell session that you start.
- A PowerShell profile is a script that runs when PowerShell starts. You can use the profile as a logon script to customize the environment.
- You can add commands, aliases, functions, variables, snap-ins, modules, and PowerShell drives. You can also add other session-specific elements to your profile so they are available in every session without having to import or re-create them.

● Powershell Profile Files

- *PowerShell supports several profiles for users and host programs. However, it does not create the profiles for you.*

Description	Path
All Users, All Hosts	Windows \$PSHOME\Profile.ps1 Linux /usr/local/microsoft/powershell/7/profile.ps1 macOS /usr/local/microsoft/powershell/7/profile.ps1
All Users, Current Host	Windows \$PSHOME\Microsoft.PowerShell_profile.ps1 Linux /usr/local/microsoft/powershell/7/Microsoft.PowerShell_profile.ps1 macOS /usr/local/microsoft/powershell/7/Microsoft.PowerShell_profile.ps1
Current User, All Hosts	Windows \$Home\Documents\PowerShell\Profile.ps1 Linux ~/.config/powershell/profile.ps1 macOS ~/.config/powershell/profile.ps1
Current user, Current Host	Windows \$Home\Documents\PowerShell\Microsoft.PowerShell_profile.ps1 Linux ~/.config/powershell/Microsoft.PowerShell_profile.ps1 macOS ~/.config/powershell/Microsoft.PowerShell_profile.ps1

○

○ Editing a Profile

- You can open any PowerShell profile in a text editor, such as Notepad.
- Here we are taking advantage of the automatic variable \$PROFILE

```
notepad $PROFILE
```

```
#To specify a specific profile.
```

```
notepad $PROFILE.AllUsersAllHosts
```

-
- Test the path of the \$PROFILE to see if it exists

```
Test-Path $profile
```

-
- It may show false because the file doesn't exist, so let's create it.

```
New-Item -Path $Profile -Type File -Force
```

●

- Now if we test it again it should show us true. Type \$PROFILE to see what the default path to the file we just created is. Navigate to it and open it up. In here, the profile file, we can make any changes that we want to happen everytime we open the console.
- We are going to add the following to our profile:

```
function prompt{
    "CYBERMAN: $($ExecutionContext.SessionState.Path.CurrentLocation) $('= ')"
}
set-alias -name list -value get-childitem
```

- To apply the changes, save the profile file, and then restart Powershell

COL

True or False: A powershell profile is a script that runs when PowerShell starts?
true

List two of the PowerShell profiles

All users/all hosts

Current user/current host

- Observing the minimal help documentation for “get-help get process” which group contains a notice to see more examples?
 - **REMARKS**
- Observing the -full help documentation of the get-command, which group contains information about piping into the cmdlet?
 - Make sure PS is updated to 5.1.19041.4170
 - **INPUTS**

MOD F01: PowerShell Fundamentals (PE1)

Note

Before Starting these practical exercises:

Run Powershell as an administrator

```
[Net.ServicePointManager]::SecurityProtocol = [Net.SecurityProtocolType]::Tls12
```

```
Update-Help -Verbose -Force -ErrorAction SilentlyContinue
```

2 The PowerShell format for cmdlets is

Verb/Noun

3 (T or F) Powershell is an open-source proprietary programming language developed by the Linux foundation.

False

4 All output and inputs from PowerShell are **.NET Objects**.

5 PowerShell methods are **Actions** that can be taken by an object.

6 Which of the following cmdlet will provide PowerShell help functionality?

Get-Help

7 I want to see examples of the cmdlet get-process. What would I type in?

Get-Help Get-Process -Examples

8 **Pipelines** are used to run many commands sequentially.

9 **Aliases** are alternate names that can be used to run commands.

10 A PowerShell object is made up of three types of data: **the object type, its properties, and its methods**.

11 The core cmdlets in PowerShell are built in **.NET Core**.

12 Cmdlet is often interchangeably called a **Command**.

13 Which of the following is a wildcard character for any string in Powershell?

"*"

14 Observing the minimal help documentation for ``get-help get-process`` which group contains a notice to see more examples?

Remarks

15 Observing the -full help documentation of `get-command`, which group contains information about piping into the cmdlet?

Inputs

16 You've been entering `gci` in PowerShell forever since you copied and pasted it from online. What is it really running as?

Get-ChildItem

16 Create an alias every time `np` is entered will open `notepad.exe`

Set-Alias -Name np -Value notepad.exe

18 An input mechanism for users to select options or provide input is called **parameters**.

19 Variables that you create at the PowerShell command line exist only while the PowerShell window is open are called **User-Created Variables**.

20 Variables that store state information for PowerShell. These variables created and maintained by Windows PowerShell are called **Automatic Variables**.

21 Customized variables that affect the PowerShell operating environment and all commands that run in the environment are called **Preference Variables**.

22 Based on the following, what would be the outcome?

```
$apple = "pears","watermelon","heap","waterfall",15,10,"squirrel"
```

```
$apple[3]
```

It will select a waterfall. It is the 4th element because indexes start at 0.

23 Based on the following, what would be the outcome?

```
$apple = "pears","watermelon","heap","waterfall",15,10,"squirrel"
```

```
$apple[-4]
```

It will select a waterfall. It is the 4th element when counting backwards. There is no index [-0] because reverse indexes start at [-1].

MOD F02: PowerShell Objects (PE2)

1 (True or False) PowerShell is an object-oriented language and shell.

True

2 (True or False) A pipeline is a series of commands connected by "|".

True

3 (True or False) Every cmdlet can be piped to any other command.

False

4 I am running powershell with administrative permissions; Based on the following:
If the following text file "test.txt" exists in \$home with the words "hello world" written inside of it, what will be the outcome if I run the following command?

Get-content \$Home\test.txt | Remove-Item

An error will occur looking for "C:\Users\student\hello world"

5 Based on the following, what is Standard-In for Where-Object?

Get-ChildItem C:\Windows\System32 | Where-Object {\$_.Name -like "bad*"}

The Standard out of Get-ChildItem

6 Based on the following, what is Standard-In for Where-Object?

Get-ChildItem C:\Windows\System32 | Where-Object {\$_.Name -like "bad*"}

The output of the pipelined series anything in system32 starting with bad

7 Based on the following "-eq Automatic " is what?

Get-Service | Where-Object StartType -eq Automatic | Select-Object -Property Name

Argument

8 From the following what is "fl"

Get-Process -id 1360 | fl * -Force

Format-List

9 Property members are **attributes** that describe an object.

10 Based on the following "Select-Object" is what?

Get-Service | Where-Object StartType -eq Automatic | Select-Object -Property Name

Statement

11 Which of the following is not a method available in get-process? (In Powershell version 5)

Get-Process | Get-Member -MemberType Method

"Strings"

12 Which of the following is not a method available in get-help?

Get-Help | Get-Member -MemberType Method

"Length"

13 Which of the following is not a method available in get-typedata?

Get-TypeData | Get-Member -MemberType Method

"Members"

14 What is the GeoID member type in Get-WinHomeLocation?

Get-WinHomeLocation | Get-Member

"Property"

15 If given the following code, what would you write in the if condition to evaluate the statement as true and output "You win!"

```
$teehee="xD"
```

```
if ($teehee -eq "xD")
```

```
{
```

```
    write-output "You win!"
```

```
}
```

MOD F03: PowerShell Functions

1 In PowerShell grouping code together and giving it a name, so that we can call it by that name later, is called **Function**.

2 How do you run a function?

Enter the name of function

3 Based on the following, what is the outcome once you run the function?

```
$arg="Talking Cat"
```

```
function hello { "Hello there $arg, how are you?" }
```

```
$arg="Schrodinger"
```

Hello there Schrodinger, how are you?

4 Developers, when making a script or function, use this to enable script users to provide input at runtime.

Parameters

5 Based on the following, what is the function name?

```
$arg="Talking Cat"
```

```
function hello { "Hello there $arg, how are you?" }
```

```
$arg="Schrodinger"
```

"Hello"

6 Missing Question

7 (True or False) Functions declared will remain if the terminal exits and reopens.

False

8 Functions you want to remain persistent should be defined in/at/with

PowerShell Profile

9 (True or False) Strings in PowerShell are always objects.

True

10 The following are two strings when running the command will _____ the strings

```
$string = "Wubba Lubba"; $string + " Dub Dub"
```

Concatenate

11 What takes a property argument passed and converts it to a string for string manipulation?

Expand Property

12 This operator will take a string occurrence and put a new one in its place.

Replace

13 What will be the output of the following?

\$YearofCovid = "Sometimes I think if all this is happening because I didn't forward that email to 10 people"

\$YearofCovid -replace "think","wonder" -replace "email","meme"

Sometimes I wonder if all this is happening because i didn't forward that meme to 10 people

14 What will be the output of the following?

"Be8azgoodhpersonxbutzdont8wasteztimetrying8tohprove it." -split {\$_ -eq "8" -or \$_ -eq "h" -or \$_ -eq 'x' -or \$_ -eq "z"}

Be a good person but don't waste time trying to prove it.

15 Based on the following, two words will be on the same line which is one of them

"Be8azgoodhpersonxbutzdont8wasteztimetrying8tohprove it." -split {\$_ -eq "8" -or \$_ -eq "h" -or \$_ -eq 'x' -or \$_ -eq "z"}

prove

16 A sequence of characters that specifies a search pattern.

Regex

17 REGEX is short for **Regular Expression**.

18 The following will result in True: 'big' -match 'b[iou]g'

True

19 The following will result in True: 'bog' -match 'b[iou]g'

True

20 The following will result in True: 'boig' -match 'b[iou]g'

False

21 The result of the following would be? (Terrible question, but answer is true.)

write-host 100

100 -match '[0-9][0-9]'

True

22 Which of the following regex character ranges will match any number?

\d

23 Which of the following regex character ranges will match any word?

\w

24 The period "." will match any character except a **newline**.

25 Which quantifier is for zero or more times?

"*"

26 Which quantifier is at least one or more times?

"+"

27 Which quantifier is for zero or one time?

"?"

28 In Powershell a variable \$message contains text. Apply a regex to validate true if it begins with the word "error".

\$message -match "^error"

29 Within the HR Employees file, count the total # of times the ": S" tag appears.

548

\$employees | Select-String ": S" -CaseSensitive | Measure-Object | Select-Object -ExpandProperty Count

Note: I simply declared \$employees = Get-Content "HR_Employee_list.txt" IOT to avoid typing Get-Content "HR_Employee_list.txt" every single time during PEs. You may see me use \$employees and Get-Content "HR_Employee_list.txt" interchangeably below.

30 Within the HR Employees file, count the total # of times "Antarctica" appears anywhere in the file.

179

Get-Content "HR_Employee_list.txt" | Select-String "Antarctica" | Measure-Object | Select-Object -ExpandProperty Count

31 Within the HR Employees file, count the total # of times the "Na" appears at the beginning of the line.

1500

Get-Content "HR_Employee_list.txt" | Select-String "^Na" | Measure-Object | Select-Object -ExpandProperty Count

32 Within the HR employees file, count the total # of times a "." appears in the file.

7874

**(\$employees -join "" -replace "[^.]", "").Length OR
(\$employees | Select-String -Pattern "\." -AllMatches).Matches.Count**

33 Count the total # of times "84" appears in the middle of a Social Security Number.
25

**Get-Content "HR_Employee_list.txt" | Select-String "-84-" | Measure-Object |
Select-Object -ExpandProperty Count**

34 Within the HR Employees file, count the total # of times two digits appear at the end of the line, using any combination of the numbers "1,2,or 3".

553

**Get-Content "HR_Employee_list.txt" | Select-String -Pattern "[123{2}\$]" |
Measure-Object**

35 Within the HR Employees file, count the total # of times "72" appears at the end of a line.

60

**Get-Content ".\HR_Employee_list.txt" | Select-String -Pattern "(72)\$" -AllMatches
| Measure-Object**

36 Within the HR Employees file, count the total # of unique area codes anywhere in the file.

716

**\$employees | Select-String -Pattern "\((\d{3})\) \d{3}-\d{4}" -AllMatches |
ForEach-Object {\$_.Matches.Groups[1].Value } | Sort-Object -Unique |
Measure-Object**

37 Within the HR Employees file, count the total # of IPs in the 150.0.0.0/8 network anywhere in the file.

11

**(\$employees | Select-String -Pattern "\b150\.(\d{1,3})\.(\d{1,3})\.(\d{1,3})\b"
-AllMatches).Count or
(Get-Content .\HR_Employee_list.txt | Select-String -Pattern
"\b150\.(\d{1,3})\.(\d{1,3})\.(\d{1,3})\b" -AllMatches).Count**

38 Within the HR Employees file, count the total # of ".mil" emails in the file.

215

**(Get-Content .\HR_Employee_list.txt | Select-String -Pattern "\b\.mil\b"
-AllMatches).Count**

39 Create a string that matches the following regex pattern `[A-Za-z]{1,3}z[0-9a-z].\:`"
Abz8X: or az0.: or xyz3?: (Basically any expression that satisfies the above condition)

40 Create a string that matches with the following pattern `"((.?.?a[0-9]){2}){2}"`
a5a7a2a3