Table of Contents

1. Introduction 3
2. Basic syntax 4
  2.1. Execution Policy 4
3. Modules 5
  3.1. Types of PowerShell Modules 5
  3.2. Key features of modules 5
  3.3. Importing modules 5
    3.3.1. -RequiredVersion 6
  3.4. Active Directory Module 6
  3.5. Finding PowerShell modules 7
4. The PowerShell help system 8
  4.1. Help-Update 8
  4.2. Finding commands (Get-Command) 8
  4.3. Using Online help 9
5. Script Creation and Execution 10
  5.1. Save the script 10
  5.2. Running the script 11
  5.3. Script output 11
    5.3.1. Format output 12
6. Naming convention 13
  6.1. Snake case 13
  6.2. Camel case 13
  6.3. Pascal case 13
  6.4. Which case do I use? 14

7. Summary and some best practice 15
7.1. Comment your code 15
7.2. Modularize Code........ 15
7.3. Test scripts 16
8. Referance 17

1. INTRODUCTION

PowerShell is a versatile and powerful scripting language and command-line shell developed by Microsoft. It was initially released in 2006 as a task automation and configuration management framework. Built on the .NET framework, PowerShell enables administrators and developers to control and automate the administration of Windows systems and applications.

PowerShell operates through cmdlets (pronounced "command-lets"), which are specialized .NET classes implementing a particular operation. These cmdlets are simple, single-function tools that manage the underlying Windows system. A key feature of PowerShell is its pipeline, where the output of one cmdlet can be passed as input to another cmdlet, allowing for complex operations to be performed with simple, concise commands.

It's essential to emphasize the interactive nature of PowerShell. When a cmdlet is run in the PowerShell console, it immediately shows the output, which makes learning and experimentation engaging (sdwheeler, 2023a).

2. BASIC SYNTAX

The basic syntax of PowerShell is straightforward, making it accessible for beginners yet powerful for experienced users. The syntax for a cmdlet is typically in the form of 'Verb-Noun', with a verb representing the action to be performed and a noun specifying the target of the action. For example, **Get-Command** retrieves a list of all available cmdlets, functions, workflows, etc., available in your PowerShell session.

PowerShell scripts are written in files with a **.ps1** extension. A fundamental aspect of PowerShell scripting is its object-oriented nature, where the output is not just text but fully-formed objects that can be manipulated.

The **Get-Help** cmdlet displays information about PowerShell cmdlets and concepts. For instance, **Get-Help Get-Command** provides detailed information about the **Get-Command** cmdlet. **Get-Service** cmdlet retrieves information about the services installed on your system. It can be used to start, stop, and manage services. **Import-Module** is the cmdlet that imports a module into your session, making the cmdlets and functions offered by the module available for use.

Understanding the basics of PowerShell involves learning how to navigate through the filesystem using cmdlets like **Set-Location** (changes the current directory) and **Get-ChildItem** (lists items in a directory). Creating and manipulating objects, and filtering and iterating through arrays of data, are also fundamental skills.

2.1. Execution Policy

A must know cmdlet is **Set-ExecutionPolicy**. Execution policies, which determine the conditions under which PowerShell loads configuration files and runs scripts, are essential for maintaining a secure environment. These policies help prevent the execution of malicious scripts. It is important to know that the -ExecutionPolicy parameter is ignored when running PowerShell on non-Windows platforms. Get-ExecutionPolicy returns **Unrestricted** on Linux and macOS. Set-ExecutionPolicy does nothing on Linux and macOS.

Use Set-ExecutionPolicy to change the execution policy.

- Restricted: No scripts can run.
- AllSigned: Only scripts signed by a trusted publisher can run.
- RemoteSigned: Downloaded scripts must be signed by a trusted publisher.
- Unrestricted: All scripts can run.

3. MODULES

PowerShell modules are an essential aspect of the PowerShell ecosystem, providing a structured way to organize and reuse scripts, functions, and cmdlets. They allow users to extend the functionality of PowerShell by adding custom commands and functionalities. Understanding the basics of PowerShell modules is crucial for anyone looking to leverage the full power of PowerShell. A PowerShell module is a package that contains PowerShell functions, cmdlets, variables, and other tools that can be added to a PowerShell session. Modules are designed to encapsulate specific functionalities, which can be easily shared and distributed (sdwheeler, 2023b).

3.1. Types of PowerShell Modules

Script Modules (.psm1) are the simplest form of modules, essentially a PowerShell script file containing functions, cmdlets, or variables that is included in the module. **Binary Module**s are compiled assemblies, typically written in a .NET language like C#. They can provide more advanced functionalities and performance improvements. **Manifest Modules (.psd1)** are manifest modules, which includes a module manifest file, which is a PowerShell data file (.psd1) that describes the contents and dependencies of the module. This file can include metadata like the author, module version, and information about which files should be processed by PowerShell.

3.2. Key features of modules

Modules allow you to organize your PowerShell scripts and functions in a way that makes them easy to reuse and share. Functions and variables in a module are executed in their own scope, which helps in avoiding naming conflicts and unintended interactions with the rest of your PowerShell environment. Modules can be versioned, allowing users to specify and use specific versions of a module.

3.3. Importing modules

The importation of modules into a PowerShell session is a straightforward process but crucial for accessing the functionalities encapsulated within these modules.

he simplest way to import a module is by using the **Import-Module** cmdlet followed by the name of the module. For example, **Import-Module -Name ModuleName**. PowerShell searches for the module in the paths specified in the **$env:PSModulePath** environment variable. If a module is not in one of the default paths, you can import it by specifying the full path to the module file. For example, **Import-Module -Name C:\Path\To\Your\Module.psm1**. PowerShell can automatically import a module when you use a command from that module, if the module is in a recognized path. This feature, available from PowerShell 3.0 onwards, simplifies the process, especially for commonly used modules (sdwheeler, u.å.-b).

3.3.1.-RequiredVersion

If you have multiple versions of a module installed, you can import a specific version by using the **-RequiredVersion** parameter. For example, **Import-Module -Name ModuleName -RequiredVersion 1.2.3**. Importing a specific version of a PowerShell module using the **-RequiredVersion** parameter is important for several reasons, particularly in environments where scripts and automations need to maintain consistency and reliability over time. Different versions of a module can contain different cmdlets, or the same cmdlets might work differently. By specifying a module version, you ensure that your scripts always use the version they were designed for, maintaining script **reliability** and **predictability**. In environments where multiple scripts interact with each other, using consistent module versions across these scripts is crucial to avoid compatibility issues. In enterprise environments, standardizing on specific module versions can help maintain a consistent and predictable environment, reducing the risk of unexpected behavior due to version discrepancies. Some organizations have strict compliance requirements that mandate the use of specific versions of software, including PowerShell modules, due to security vetting. n sensitive environments, updates to software, including modules, are often controlled and gradual. Using **-RequiredVersion** helps maintain control over when and how new versions are adopted.

3.4. Active Directory Module

The Active Directory (AD) module for PowerShell is a powerful tool for managing AD resources. It includes cmdlets for creating, modifying, and removing AD objects like users, groups, and computers. To use the AD module, you first need to import it into your PowerShell session with Import-Module -Name ActiveDirectory. This command makes the AD cmdlets available in your session. After importing the module, you can use its cmdlets to perform various tasks. For example, Get-ADUser retrieves information about AD user accounts, and New-ADUser creates new user accounts. If you don’t know what Active Directory is, don’t worry, we will cover that in its own lecture.

3.5. Finding PowerShell modules

Discovering available PowerShell modules, whether installed on your system or available online, is essential for extending your toolkit. The **Get-Module -ListAvailable** cmdlet lists all modules installed on your system. This command shows modules in the paths included in the **$env:PSModulePath** environment variable. The **Find-Module** cmdlet is used to search for modules in online repositories like the PowerShell Gallery. For example, **Find-Module -Name *ActiveDirectory*** searches for modules related to Active Directory in the PowerShell Gallery.

The PowerShell Gallery (https://www.powershellgallery.com/) is the central repository and the most common source for finding and downloading new modules. You can browse this website to find modules for various tasks and functionalities.

When you find a module you wish to use from an online source, you can install it using **Install-Module -Name ModuleName**. This cmdlet downloads and installs the module from the PowerShell Gallery or other configured repositories.

It is important to ensure the module is compatible with your version of PowerShell and your operating system. Before using a module, especially third-party ones, read its documentation to understand its functionalities and any potential issues or dependencies. Keep your modules updated for the latest features and security fixes. The **Update-Module** cmdlet can be used to update installed modules.

4. THE POWERSHELL HELP SYSTEM

The PowerShell help system is a comprehensive and integral feature that guides users through the vast landscape of cmdlets, functions, modules, and scripts. It's designed to provide detailed information about these components, including their usage, parameters, examples, and much more.

The primary way to access help in PowerShell is through the **Get-Help** cmdlet. For example, to get help on the **Get-Process** cmdlet, you would use **Get-Help Get-Process**. This command displays basic information about the **Get-Process** cmdlet, including a brief description, syntax, related links, and more. To get more detailed information, including parameters, inputs, outputs, and detailed examples, you can use the **-Detailed** switch, **Get-Help Get-Process -Detailed**. If you're specifically looking for examples of how to use a cmdlet, you can use the **-Examples** switch, **Get-Help Get-Process -Examples**. This command provides various examples of how to use **Get-Process**, which is particularly useful for learning how to apply the cmdlet in different scenarios.

To get information about a specific parameter of a cmdlet, use the **-Parameter** switch, **Get-Help Get-Process -Parameter Name**.

4.1. Help-Update

In PowerShell, **Update-Help** is a cmdlet that plays a crucial role in maintaining the usefulness and relevance of the help system. The help system in PowerShell is designed to provide detailed, contextual documentation for cmdlets, functions, concepts, and more, aiding users in understanding and effectively using PowerShell's capabilities.

4.2. Finding commands (Get-Command)

If you're unsure about the exact cmdlet you need, you can use the **Get-Command**, **Get-Command *process***. This command lists all cmdlets, functions, aliases, etc., that include the word "process" in their names.

Imagine you want to write a script that involves manipulating services on a computer. You are not completely familiar with the cmdlets related to services. Here's one approach on how you could use the help system. First you are trying to find a service-related cmdlets:

```
Get-Command *service*
```

This lists cmdlets like Get-Service, Start-Service, Stop-Service, etc. Now you need to learning how to use Get-Service.

```
Get-Help Get-Service -Detailed
```

This provides detailed information on **Get-Service**, including its parameters, syntax, and examples.

If you figure out that it is, **Stop-Service** that you are looking for, you might need to get some examples for Stop-Service.

```
Get-Help Stop-Service -Examples
```

This shows a small, but practical, examples of how to use **Stop-Service**, which can be very helpful for scripting.

In conclusion, the PowerShell help system is a rich, user-friendly resource that empowers users to understand and effectively utilize the capabilities of PowerShell. By providing detailed documentation, examples, and online resources, it serves as an invaluable tool for both beginners and experienced PowerShell users. Through regular interaction with this system, users can greatly enhance their PowerShell proficiency and scripting expertise (sdwheeler, u.å.-a).

4.3. Using Online help

Sometimes, it might be more convenient to read help documentation online. You can use the **-Online** switch to open the online version of the help file in your default web browser, **Get-Help Get-Process -Online**

5. SCRIPT CREATION AND EXECUTION

PowerShell scripts are essentially text files containing a sequence of PowerShell commands, saved with a **.ps1** file extension. Let's try to break down the key aspects of script creation and execution. You can write PowerShell scripts in any text editor (like Notepad), but using an editor with PowerShell syntax support, like Visual Studio Code or PowerShell Integrated Scripting Environment (ISE), is beneficial. These editors provide syntax highlighting, command completion, and other helpful features. Like mention above, a PowerShell script is a series of PowerShell commands, but scripts can also include functions, loops, conditionals, error handling, and more, similar to other scripting languages. Here's a simple script that lists all processes on a computer:

```
Get-Process | Sort-Object CPU -Descending | Select-Object -First 10
```

This script retrieves all running processes, sorts them by CPU usage in descending order, and then selects the top 10.

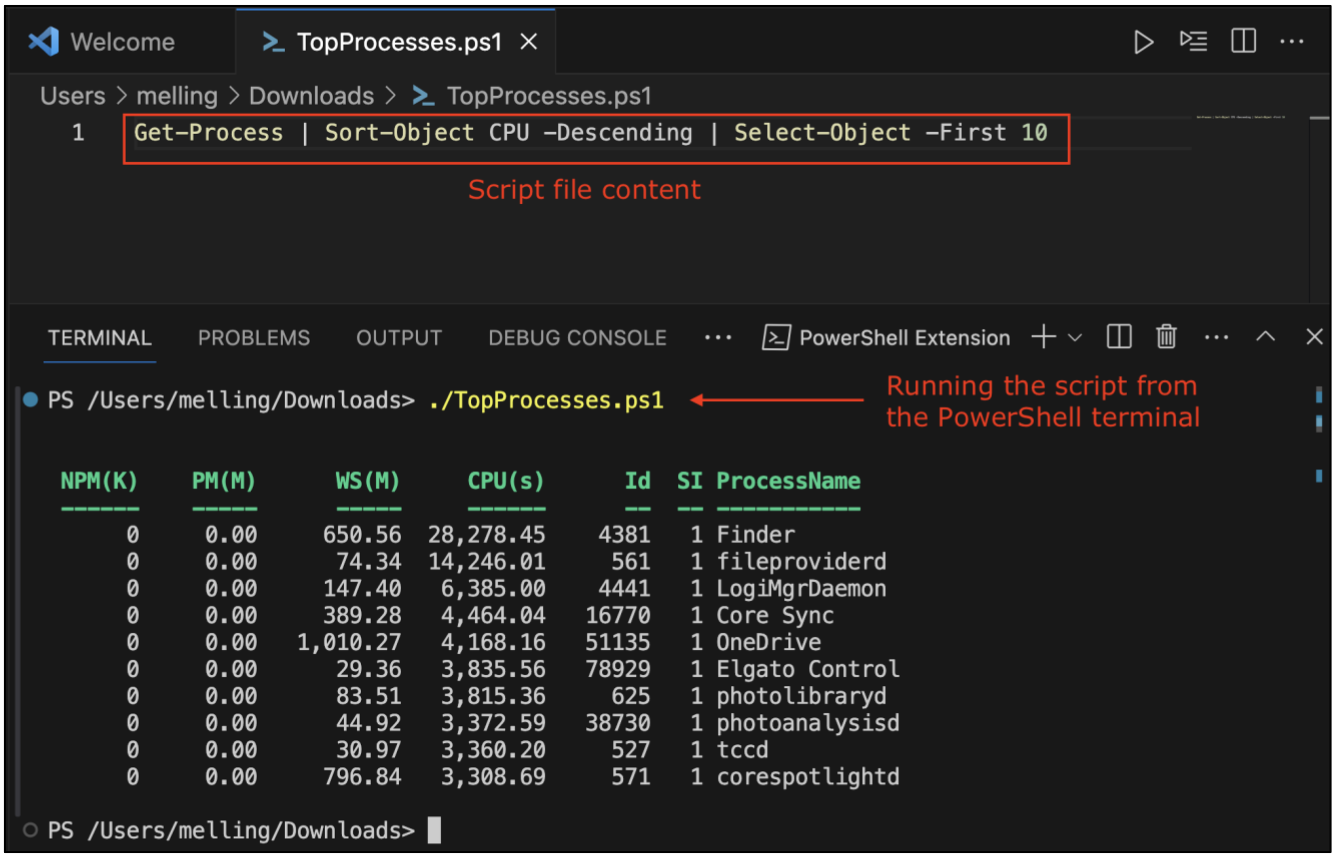
```
PS /Users/melling> Get-Process | Sort-Object CPU -Descending | Select-Object -First 10

 NPM(K)    PM(M)      WS(M)     CPU(s)      Id  SI ProcessName
 ------    -----      -----     ------      --  -- -----------
      0     0.00     636.20  28,246.38    4381   1 Finder
      0     0.00      76.39  14,091.17     561   1 fileproviderd
      0     0.00     147.07   6,374.23    4441   1 LogiMgrDaemon
      0     0.00     407.91   4,462.08   16770   1 Core Sync
      0     0.00   1,329.66   3,944.28   51135   1 OneDrive
      0     0.00      29.55   3,834.45   78929   1 Elgato Control
      0     0.00      99.78   3,815.36     625   1 photolibraryd
      0     0.00      55.05   3,372.50   38730   1 photoanalysisd
      0     0.00      31.05   3,359.99     527   1 tccd
      0     0.00     820.31   3,296.51     571   1 corespotlightd
```

5.1. Save the script

Save the script with a **.ps1** file extension. For example, **TopProcesses.ps1**. Save the script in a location where you can easily access it from the PowerShell prompt. PowerShell's execution policies are a security feature that controls the conditions under which PowerShell loads configuration files and runs scripts. Use **Get-ExecutionPolicy** to check the current execution policy on your system.

5.2. Running the script

From the PowerShell prompt, you can navigate to the directory containing the script and run it by typing .\ScriptName.ps1. If your script is designed to accept parameters, you can pass them while invoking the script. For example, if the TopProcesses.ps1 accepted a parameter for the number of processes to display, you could run it as .\TopProcesses.ps1 -Number 5. In the example below, we have not designed TopProcesses to accept parameters.



```
Get-Process | Sort-Object CPU -Descending | Select-Object -First 10
```

```
PS /Users/melling/Downloads> ./TopProcesses.ps1

 NPM(K)    PM(M)      WS(M)     CPU(s)      Id  SI ProcessName
 ------    -----      -----     ------      --  -- -----------
      0     0.00     650.56  28,278.45    4381   1 Finder
      0     0.00      74.34  14,246.01     561   1 fileproviderd
      0     0.00     147.40   6,385.00    4441   1 LogiMgrDaemon
      0     0.00     389.28   4,464.04   16770   1 Core Sync
      0     0.00   1,010.27   4,168.16   51135   1 OneDrive
      0     0.00      29.36   3,835.56   78929   1 Elgato Control
      0     0.00      83.51   3,815.36     625   1 photolibraryd
      0     0.00      44.92   3,372.59   38730   1 photoanalysisd
      0     0.00      30.97   3,360.20     527   1 tccd
      0     0.00     796.84   3,308.69     571   1 corespotlightd

PS /Users/melling/Downloads>
```

Figure 1 - VS Code, script content and terminal

5.3. Script output

Script output in PowerShell is an important aspect to understand, as it determines how the results of your scripts are presented and utilized. PowerShell is designed to work with objects, and the way it handles script output reflects this object-oriented nature. By default, the output of a PowerShell script is displayed in the console (or PowerShell window). When a cmdlet or a statement in a script produces output, it is sent directly to the console unless otherwise directed.

PowerShell outputs objects, not just text. This means that the output can be manipulated and processed further as structured data, which is a key strength of PowerShell.

You can redirect the output of a script to a file using redirection operators. For instance, **.\YourScript.ps1 > Output.txt** redirects the standard output to **Output.txt**. If you want to append the output to an existing file, you can use **>>** instead of **>**.

Error messages can be redirected separately using the error stream redirection operator **2>**. For example, **.\YourScript.ps1 2> ErrorLog.txt** redirects only the error output. PowerShell supports multiple output streams (such as output, error, warning, verbose, and debug). Each can be redirected separately, providing fine-grained control over script output.

5.3.1.Format output

Cmdlets like **Format-Table**, **Format-List**, **Format-Wide**, etc., are used to format the output for display. These are particularly useful when you want to present data in a specific layout or format in the console. Formatting cmdlets should be used at the end of the pipeline or before the output is displayed. When output is redirected to a file, using formatting cmdlets might result in loss of object structure, as these cmdlets convert objects to formatted text.

You can capture the output of a script or command in a variable. For example, **$result = .\YourScript.ps1** captures the output of **YourScript.ps1** in the **$result** variable. If you need to capture the output in both a file and a variable simultaneously, you can use a tee-like operation with **Tee-Object**. For example, **.\YourScript.ps1 | Tee-Object -FilePath Output.txt** (sdwheeler, u.å.-c).

Use **return** to explicitly send output from a function or script. However, in PowerShell, any expression or command that produces output and is not captured or redirected will inherently be part of the script's output.

To suppress output, you can redirect it to **$null**, for example, **$output = Get-Process > $null**.

Understanding and effectively managing script output is crucial for both interpreting the results of PowerShell scripts and for controlling how and where these results are presented or stored. Whether you're displaying output in the console, redirecting it to files, or capturing it for further processing, PowerShell offers a flexible and powerful system for handling output. This system's versatility, especially its ability to work with structured objects, sets PowerShell apart from traditional shell scripting and enhances its capabilities for advanced scripting and automation tasks.

6. NAMING CONVENTION

Snake case, camel case, and Pascal case are naming conventions used in programming and scripting for variable names, function names, and other identifiers. Each has its specific format and use cases. The choice of naming convention often depends on the programming language and the conventions established by its community or frameworks. For example, Python developers often use snake case, while Java and JavaScript developers typically use camel case for variables and functions. Snake case can sometimes be easier to read, making it a good choice for identifiers with multiple words. However, it might take up more space and can be less visually appealing in some contexts. In some programming environments, different cases are used to signify different types of entities. For example, in C#, it’s common to use Pascal case for methods and public members, and camel case for local variables and parameters (*Snake Case VS Camel Case VS Pascal Case VS Kebab Case – What’s the Difference Between Casings?*, 2022).

6.1. Snake case

In snake case, words are all lowercase and separated by underscores (_). For example: **employee_name**, **total_income**. Snake case is commonly used in languages like Python and Ruby. It's preferred for its readability, especially in situations where case sensitivity is significant or in systems that are case-insensitive.

6.2. Camel case

Camel case starts with a lowercase letter and then uses uppercase letters to start new words. Example: **employeeName**, **totalIncome**. It's widely used in programming languages like JavaScript, Java, and C#. Camel case is typically preferred for naming variables and functions in these languages due to its compactness and the convention established by the language's community.

6.3. Pascal case

Pascal case is similar to camel case but starts with an uppercase letter. Each word, including the first, starts with a capital letter. Example: **EmployeeName**, **TotalIncome**. It is commonly used for class names in object-oriented programming, namespaces, and methods in languages like C#, Java, and C++. The use of Pascal case in these situations helps distinguish class names and methods from variable names.

6.4. Which case do I use?

In PowerShell, the naming conventions generally follow a mixed approach, but certain patterns are commonly observed and recommended within the PowerShell community and in official PowerShell documentation. These conventions help in writing code that is consistent, readable, and aligns with the practices of the broader PowerShell user base.

PowerShell cmdlets use Pascal Case and follow a 'Verb-Noun' naming convention. The verb is a standard verb approved by PowerShell (like **Get**, **Set**, **Remove**), and the noun is typically Pascal Case, **Get-Process**, **Set-ExecutionPolicy**.

Function names in PowerShell scripts and modules also typically follow the Pascal Case convention, adhering to the 'Verb-Noun' format. This consistency with cmdlet naming makes functions look and feel like native cmdlets, for example: **Convert-ToCsv**, **Invoke-DataCleanup**.

For variable names, camel case is commonly used. This distinction between variable names and function/cmdlet names helps in easily differentiating variables from commands in scripts, **$employeeList**, **$currentDate**.

Parameters within cmdlets and functions usually follow camel case. This is consistent with variable naming and helps maintain readability, and it could look something like this: **param($filePath, $numberOfRetries)**.

To summarize, PowerShell scripting, adhering to the recommended naming conventions of using Pascal Case for cmdlets and function names, and camel case for variables and parameters, is a best practice that enhances the readability, maintainability, and community alignment of your scripts. These conventions provide a clear distinction between different types of script elements and ensure consistency across your PowerShell codebase.

7. SUMMARY AND SOME BEST PRACTICE

Adhering to best practices in scripting, particularly in a powerful environment like PowerShell, is crucial for creating efficient, maintainable, and reliable scripts.

7.1. Comment your code

Properly commenting your code is vital for readability and maintainability, especially when others (or even your future self) need to understand or modify the script. Comments should clearly explain what the code is doing, why certain choices were made, and how the components work together. This is especially important for complex logic or unusual solutions. Use inline comments sparingly and only where necessary to explain complex parts of the code. Avoid stating the obvious, which can clutter the code.

PowerShell supports comment-based help, which allows you to embed a script's documentation within the script itself. This can be accessed using the **Get-Help** cmdlet. It should include a synopsis, description, parameter explanations, examples, and any additional notes.

```
<#
.SYNOPSIS
Brief description of the script.

.DESCRIPTION
A more detailed description of the script and its functionality.

.PARAMETER Param1
Description of the parameter.

.EXAMPLE
PS> .\YourScript.ps1 -Param1 "Value"
An example of how to run the script.
#>
```

7.2. Modularize Code

Breaking your script into smaller, modular components not only makes it more readable but also enhances reusability and simplifies testing. Write functions for reusable code blocks. Each function should accomplish a single task or a closely related set of tasks. Ensure functions are self-contained, with clearly defined inputs (parameters) and outputs (return values), and minimal dependencies on external variables. Use clear and consistent naming conventions for functions and variables to indicate their purpose and scope.

7.3. Test scripts

Testing is crucial to ensure your script works as intended and handles errors gracefully. For complex scripts, especially those that are part of a larger project, consider using a unit testing framework like Pester. Pester allows you to write tests that validate the behavior of your PowerShell code.

Include proper error handling using try-catch blocks. This ensures that your script can handle unexpected situations and provide meaningful error messages. It is also important to check and validate inputs, especially if your script takes parameters or user input. Ensure that the script behaves correctly with both valid and invalid inputs. And last, but not least, implement logging within your script to capture its flow and any errors. This is invaluable for debugging and understanding script behavior in production.

8. REFERANCE

sdwheeler. (u.å.-a). *Get-Command (Microsoft.PowerShell.Core)—PowerShell*. Hentet 20. desember 2023, fra https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.core/get-command?view=powershell-7.4

sdwheeler. (u.å.-b). *Import-Module (Microsoft.PowerShell.Core)—PowerShell*. Hentet 20. desember 2023, fra https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.core/import-module?view=powershell-7.4

sdwheeler. (u.å.-c). *Tee-Object (Microsoft.PowerShell.Utility)—PowerShell*. Hentet 21. desember 2023, fra https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.utility/tee-object?view=powershell-7.4

sdwheeler. (2023a, september 26). *How to use the PowerShell documentation—PowerShell*. https://learn.microsoft.com/en-us/powershell/scripting/how-to-use-docs?view=powershell-7.4

sdwheeler. (2023b, desember 7). *about Modules—PowerShell*. https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_modules?view=powershell-7.4

Snake Case VS Camel Case VS Pascal Case VS Kebab Case – What's the Difference Between Casings? (2022, november 29). FreeCodeCamp.Org. https://www.freecodecamp.org/news/snake-case-vs-camel-case-vs-pascal-case-vs-kebab-case-whats-the-difference/