**NAME:** - ANSHUL SAINI  
**MAIL:** - [sainianshul7565@gmail.com](mailto:sainianshul7565@gmail.com)

**User ID**: - 34732

**Date: -** 13/08/2025

**Batch: -** DWS\_B5\_25VID2550

**Topic Assignment: -**

**Basic Powershell Cmdlets**

* Process check and Delete
* For Loop
* Set-Ini Value in psadt
* How to catch return Codes with in PowerShell
* Individual Command logging within Script in psadt
* MSIX Technology Fundamentals with Architecture
* MSIX Benefits and Why MSIX

**Basic Powershell Cmdlets**

* **Process check and Delete: -**

**Get-Process: -** The Get-Process cmdlet retrieves details about processes running on a Windows system. You can query processes by name, ID, or without parameters to list all processes. It’s useful for monitoring applications, troubleshooting, or verifying if a program is active before performing actions like termination or resource management.

**Filtering Get-Process: -** Filtering with parameters like -Name or -Id narrows results to specific processes. For example, Get-Process -Id 1234 targets one process by ID, making it easier to manage or monitor. Filtering helps in precise identification, especially in automation scripts where stopping or checking the wrong process can cause issues.

**Checking if Process is Running: -** Using if (Get-Process -Name "myProcess.exe") allows you to determine if a specific application is active. If it returns output, the process exists; otherwise, it’s not running. This check is essential for automation workflows to prevent duplicate instances or to conditionally start or stop services safely.

if (Get-Process -Name "Telegram") {

Write-Host "Process myProcess.exe is running"

} else {

Write-Host "Process myProcess.exe is not running"

}

**Stop-Process: -** Stop-Process ends a running process immediately, using either its name or process ID. For example, Stop-Process -Id 1234 terminates a specific task. This is helpful for closing unresponsive applications or freeing system resources, but must be used carefully to avoid disrupting critical system functions or unsaved work.

**Specifying Process Name or ID in Stop-Process: -** You can end processes by name (Stop-Process -Name "chrome.exe") to close all instances, or by ID to terminate only a single targeted process. This flexibility supports both general cleanup tasks and precise control during troubleshooting or deployment scripts in environments like PSAppDeployToolkit.

**if (Get-Process -Name "Telegram") {**

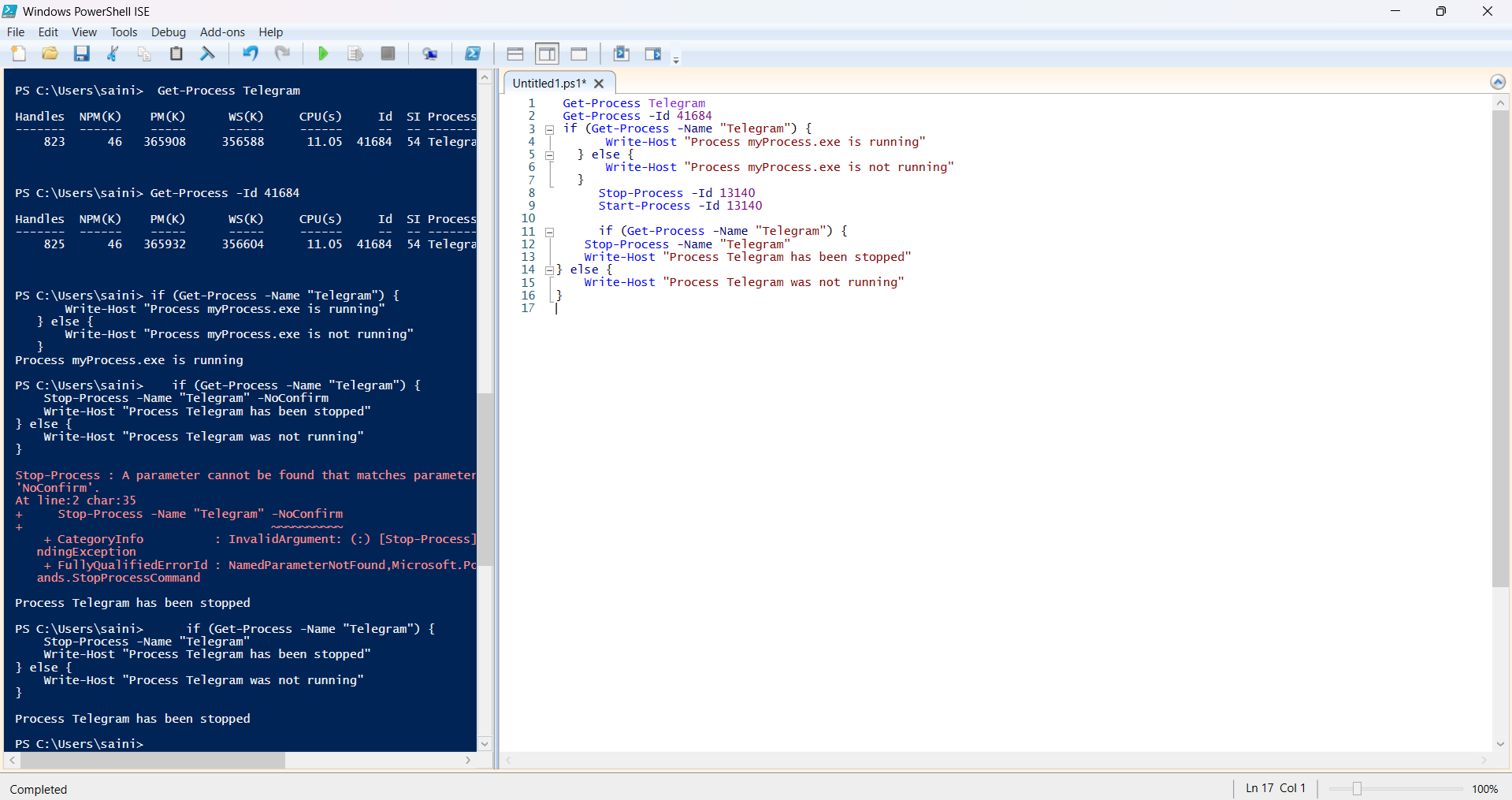
**Stop-Process -Name "Telegram"**

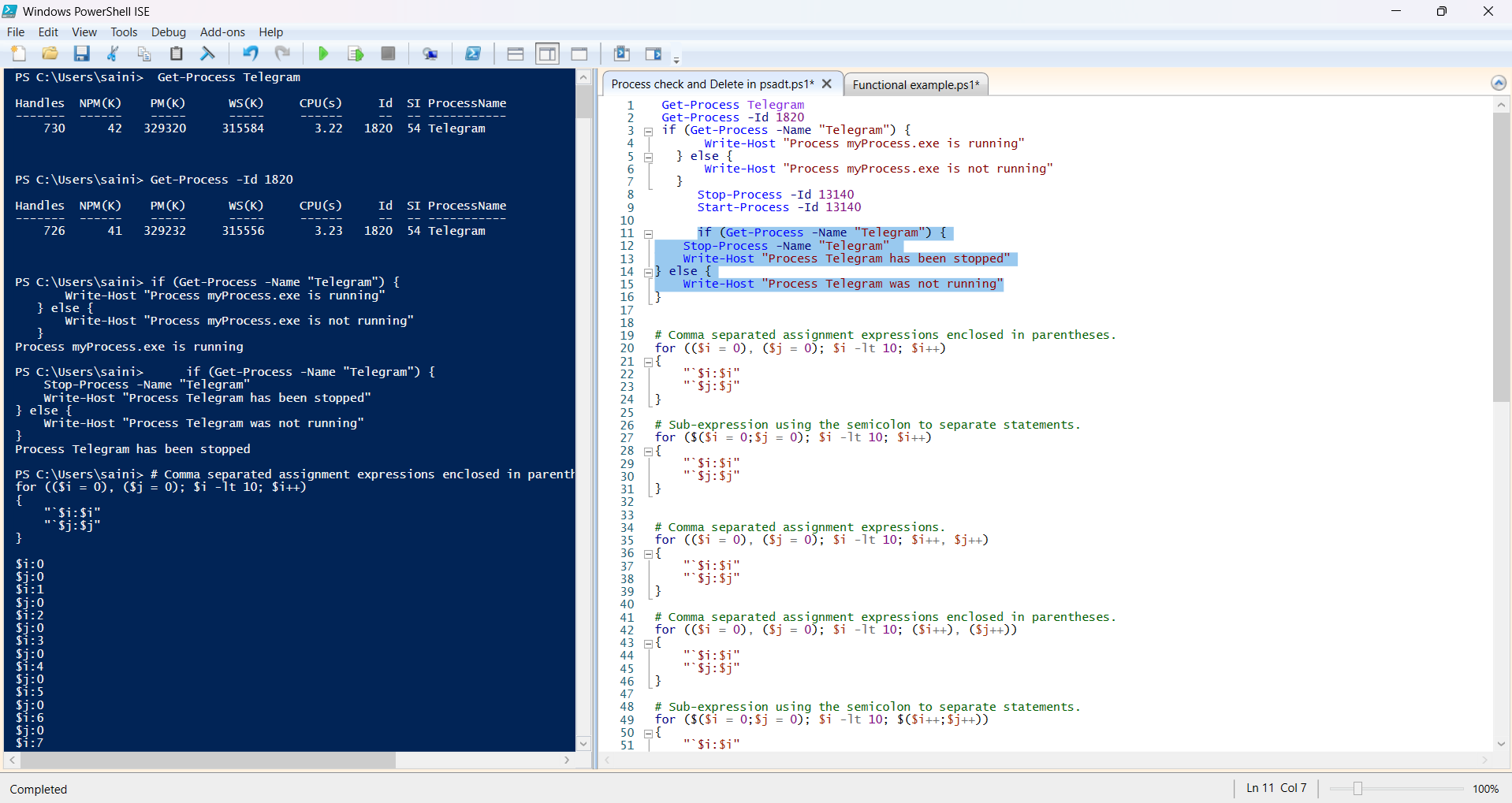
**Write-Host "Process Telegram has been stopped"**

**} else {**

**Write-Host "Process Telegram was not running"**

**Practice Activity: -**

****

****

* **For Loop: -**

The for statement (also known as a for loop) is a language construct you can use to create a loop that runs commands in a command block while a specified condition evaluates to $true.

A typical use of the for loop is to iterate an array of values and to operate on a subset of these values. In most cases, if you want to iterate all the values in an array, consider using a foreach statement.

**Some Programs: -**

**# Comma separated assignment expressions enclosed in parentheses.**

for (($i = 0), ($j = 0); $i -lt 10; $i++)

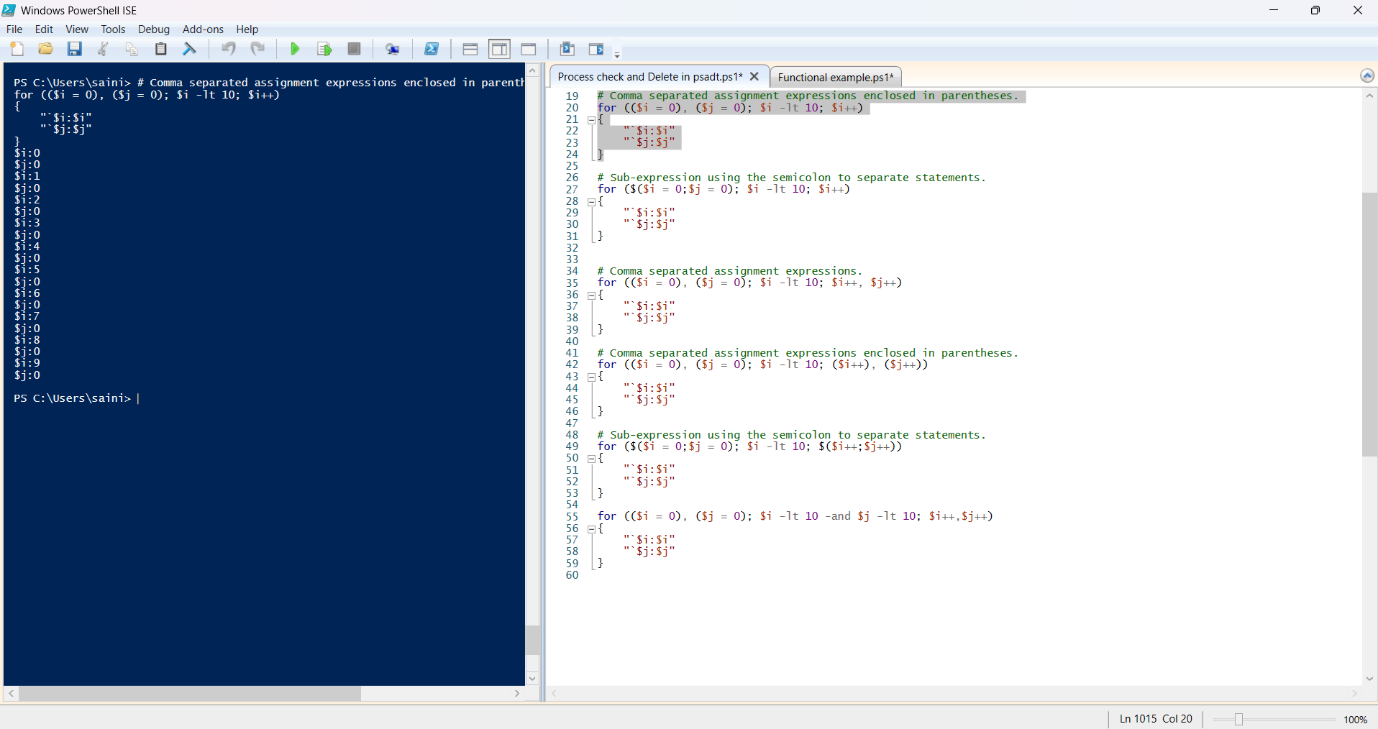
{

"`$i:$i"

"`$j:$j"

}

**Practice Activity: -**

****

**# Sub-expression using the semicolon to separate statements.**

for ($($i = 0;$j = 0); $i -lt 10; $i++)

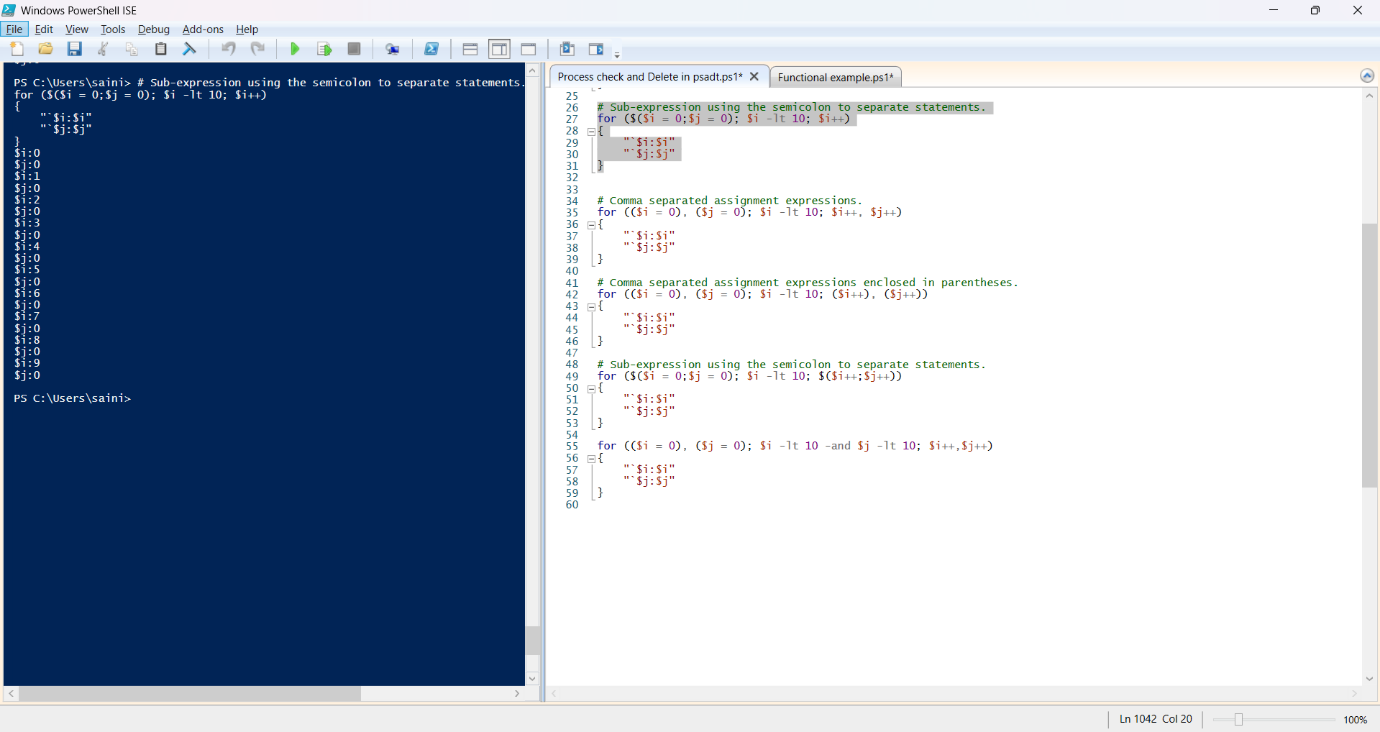
{

"`$i:$i"

"`$j:$j"

}

**Practice Activity: -**



**# Comma separated assignment expressions.**

for (($i = 0), ($j = 0); $i -lt 10; $i++, $j++)

{

"`$i:$i"

"`$j:$j"

}

**# Comma separated assignment expressions enclosed in parentheses.**

for (($i = 0), ($j = 0); $i -lt 10; ($i++), ($j++))

{

"`$i:$i"

"`$j:$j"

}

**# Sub-expression using the semicolon to separate statements.**

for ($($i = 0;$j = 0); $i -lt 10; $($i++;$j++))

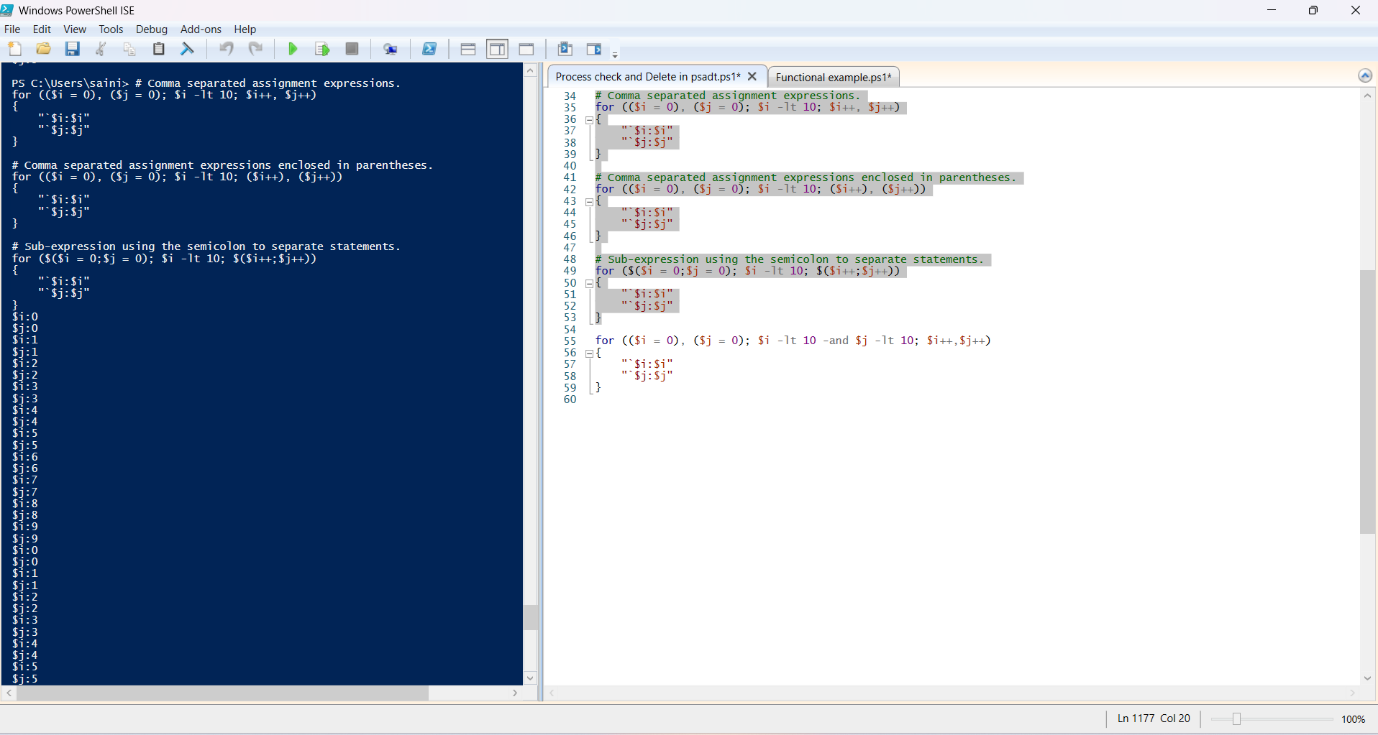
{

"`$i:$i"

"`$j:$j"

}

**Practice Activity: -**



For multiple **Conditions** use logical operators as demonstrated by the following example.

for (($i = 0), ($j = 0); $i -lt 10 -and $j -lt 10; $i++,$j++)

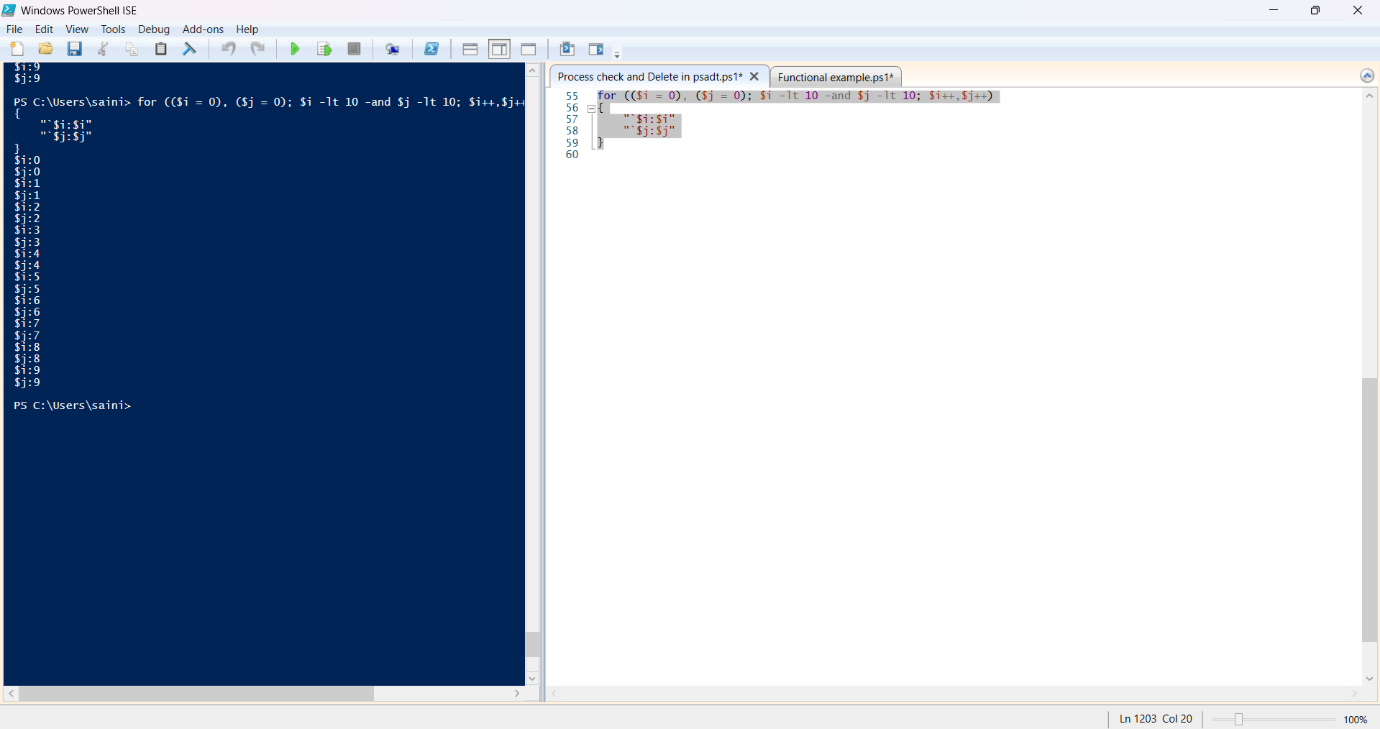
{

"`$i:$i"

"`$j:$j"

}

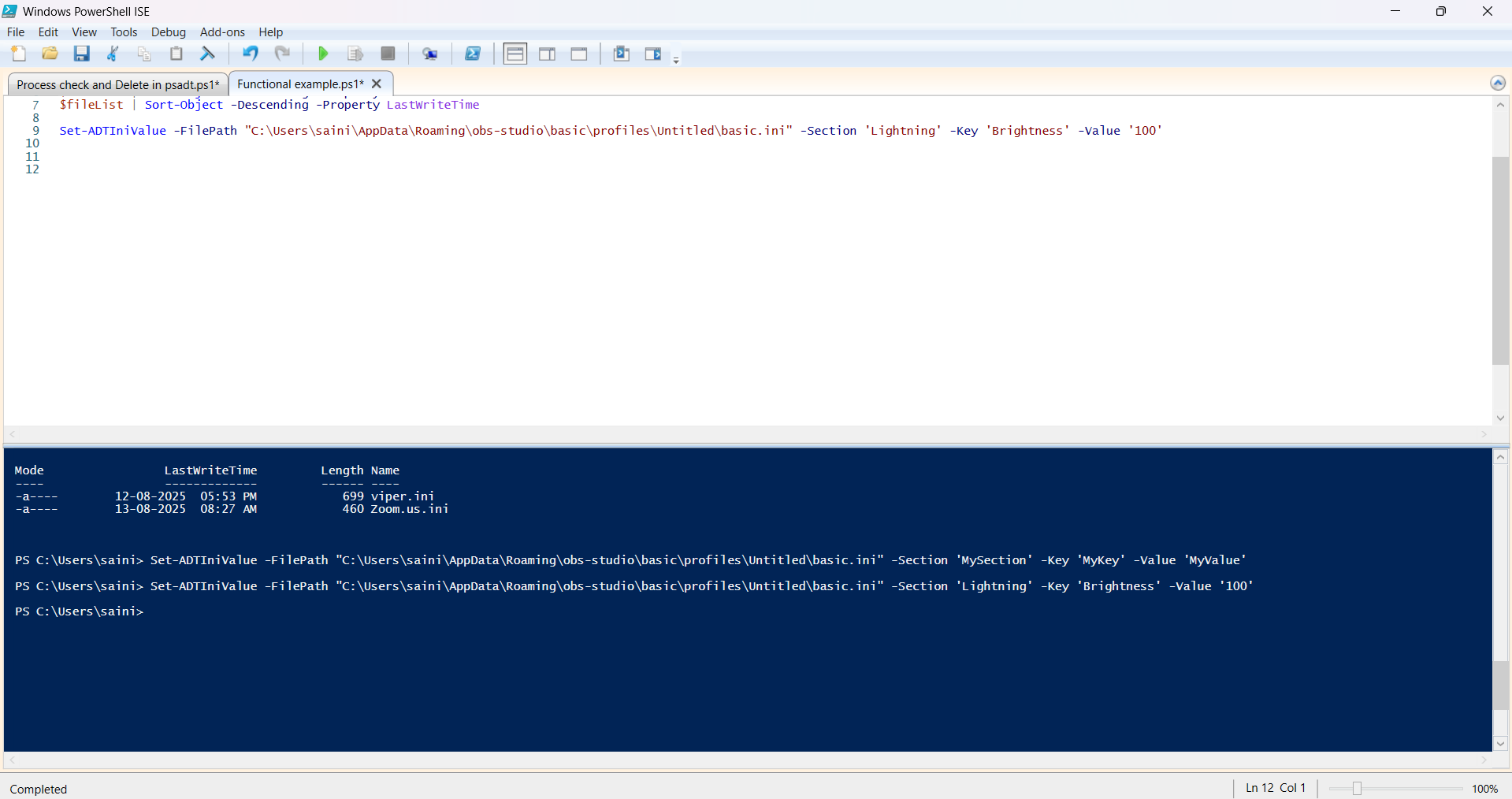
**Practice Activity: -**

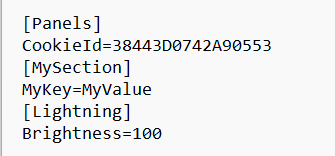
****

* **Set-Ini Value in psadt: -**

The PSAppDeployToolkit's Set-ADTIniValue function allows you to set a value within a specified section and key in an INI file. The function takes the path to the INI file, the section, the key, and the value as parameters.

Set-ADTIniValue -FilePath "C:\Users\saini\AppData\Roaming\obs-studio\basic\profiles\Untitled\basic.ini" -Section 'Lightning' -Key 'Brightness' -Value '100'

**Practice Activity: -**



* **How to catch return Codes with in PowerShell: -**

1. **$LASTEXITCODE**

This automatic variable stores the exit code from the last external program run in PowerShell. A 0 means success; non-zero means error.

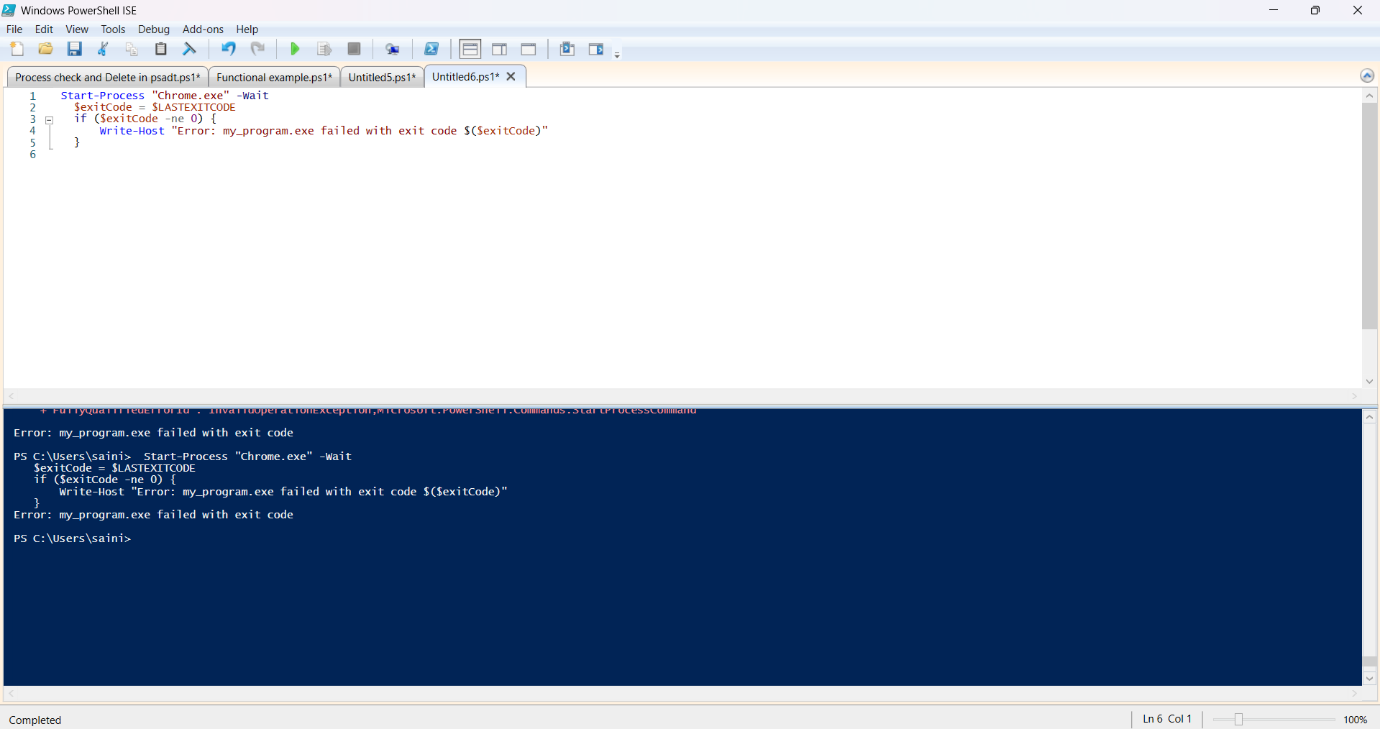
Start-Process "Chrome.exe" -Wait

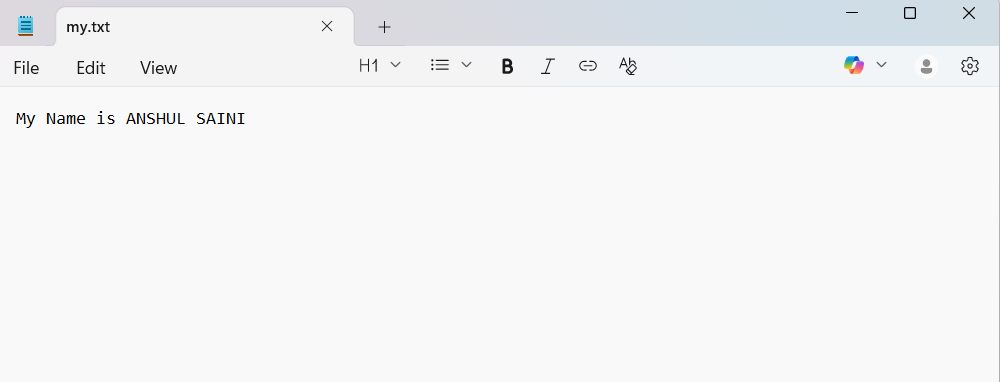
$exitCode = $LASTEXITCODE

if ($exitCode -ne 0) {

Write-Host "my\_program.exe failed with exit code $($exitCode)"

}

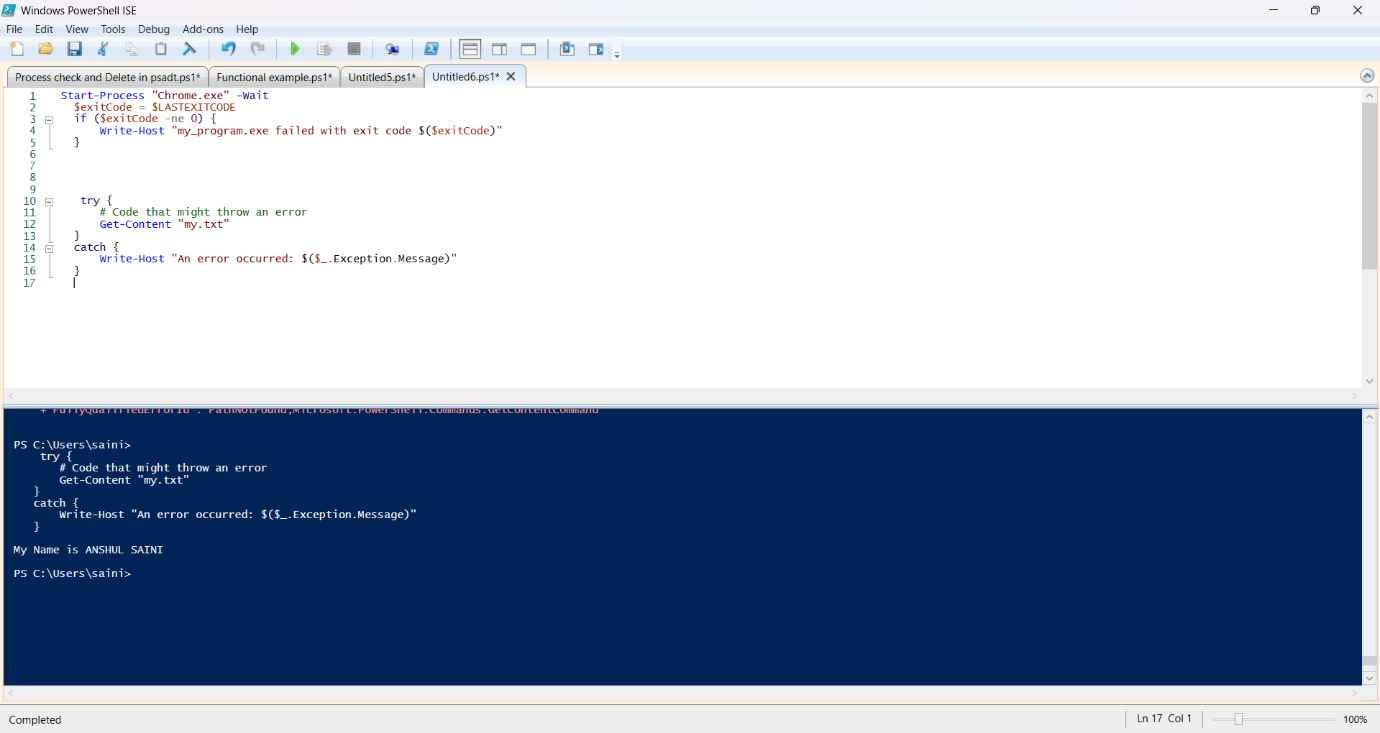
**Practice Activity: -**



1. **try...catch**

try encloses commands that may fail; catch runs if an error occurs.

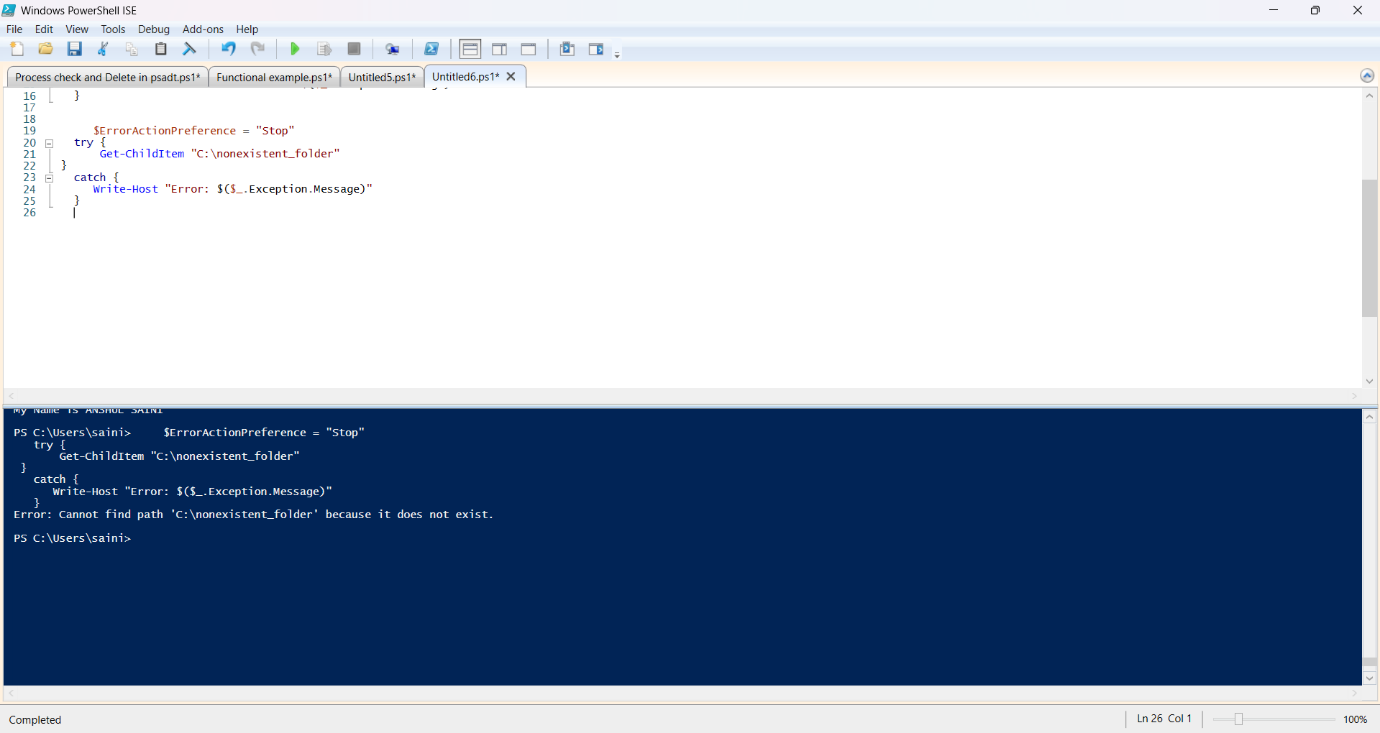
**Practice Activity: -**



1. **$ErrorActionPreference**

This variable controls error handling behavior. Setting it to "Stop" makes all non-terminating errors act as terminating, enabling try...catch capture.

**Practice Activity: -**



* **Individual Command logging within Script in psadt: -**

In PSADT, you can log individual commands and actions using the Write-Log function, which writes custom messages to the main toolkit log.

## Example of logging a message

    Write-Log "Starting the installation of Application X."

    ## Example of logging a command execution

    Execute-Process -Path "msiexec.exe" -Parameters "/i application.msi /qn"

    Write-Log "MSI installation command executed."

    ## Example of logging a status update

    If (Test-Path -Path "C:\Program Files\Application X") {

        Write-Log "Application X installation successful."

    } Else {

        Write-Log -Message "Application X installation failed." -Severity "Error"

    }

* **MSIX Technology Fundamentals with Architecture: -**

**MSIX Overview**  
MSIX is a modern Windows application packaging format, combining the best of MSI, AppX, App-V, and ClickOnce. It supports both desktop and Microsoft Store apps, offering enhanced security, efficient updates, and a clean installation/uninstallation process, minimizing system clutter compared to older installers while maintaining compatibility with Windows deployment tools.

**Architecture – Package Structure**  
An MSIX package is essentially a ZIP archive containing application binaries, configuration files (XML), and optional assets like icons and manifest files. It uses containerization, running apps in isolated environments for better security and stability, preventing conflicts with the OS, other applications, and user profiles.

**Deployment Methods**  
MSIX packages can be deployed using Microsoft Intune, MECM, or directly via the Microsoft Store. This flexibility supports both enterprise and consumer distribution scenarios, allowing organizations to manage deployments efficiently while leveraging cloud or on-premises infrastructure for controlled, secure, and scalable application delivery.

**Key Features and Benefits**  
MSIX ensures signed and validated packages, preventing tampering. It supports efficient differential updates, reducing bandwidth and deployment time. Installations and removals are clean, with minimal registry residue. It works for both Win32 and UWP apps, and supports App Attach for dynamic application provisioning and management in virtualized environments.

**MSIX vs MSI**  
Compared to MSI, MSIX offers stronger security via signing and validation, more efficient update mechanisms, streamlined deployment processes, and container-based application isolation. These enhancements improve security, stability, and manageability while reducing operational overhead, making MSIX a better fit for modern Windows environments requiring flexibility and efficiency.

* **MSIX Benefits: -**

MSIX is a modern Windows app packaging format that offers many benefits. It provides highly reliable installations with a 99.96% success rate and ensures clean uninstalls, reducing system issues. It saves network bandwidth by downloading only necessary data blocks and optimizes disk space by sharing common files. Updates are seamless, avoiding full reinstalls. MSIX enhances security by running apps in isolated containers and supports streamlined deployment with automation, remote installs, and CI/CD integration. It works with both traditional and modern apps, adds features like containerization and version control, and reduces the need for repackaging, making deployment easier for IT teams.

* **Why MSIX: -**

MSIX is the modern packaging format for Windows, built to work seamlessly with future Windows updates and features. It overcomes problems found in older formats, such as file duplication, complex updates, and system instability. With streamlined installation, fast updates, and efficient resource use, it improves the user experience. MSIX supports modern technologies like containerization, making it perfect for creating and deploying current Windows applications. It also integrates with management tools like Group Policy for centralized control in enterprises. By offering security, efficiency, and easier management, MSIX is set to transform how applications are deployed for both developers and IT teams.