Advanced Scripting   
Profiles, Functions

Last Updated: 1/24/2024 2:41 PM Version 1  
Document Prepared for: CYBER360 Student

# Name Click here to enter name ID Click here to enter id

# Instructions

Answer all questions directly in this document. You will save and upload this completed document as your homework submission.

# Overview

A few short exercises to explore profile scripts and create scripted functions.

# Setup

## Requirements

* VSCode
* Windows PowerShell (Desktop edition, version 5)
* PowerShell Core for Windows
* PowerShell Core for Linux (or macOS)

# Task 1—Profile Paths

Think of *profiles* as startup scripts for interactive PowerShell sessions.

## Steps

1. Launch a session for each of Windows PowerShell Desktop, PowerShell Core for Windows, and PowerShell Core for Linux. In each, enter **$PROFILE** to see the path to *your* profile on *your* computer (e.g. current user and host). You will see different paths under different PowerShell editions and versions. Record the value of your **$PROFILE** for each:
   1. Windows PowerShell Desktop Click or tap here to enter text.
   2. PowerShell Core for Windows Click or tap here to enter text.
   3. PowerShell Core for Linux (or macOS) Click or tap here to enter text.
2. Enter **$PROFILE | fl -force** to see *all four* profile paths. Try this at each of your three PowerShell prompts. Is any path a duplicate of another, or are all twelve paths distinct? Click or tap here to enter text.
3. Let’s see which if any of those profile scripts actually exist on your computer:

$props = "AllUsersAllHosts", "AllUsersCurrentHost"

$props += "CurrentUserAllHosts","CurrentUserCurrentHost"

foreach ($p in $props) {

"$($p): $($PROFILE.$p): $(Test-Path $PROFILE.$p)"

}

* 1. Let’s unpack that command sequence. The first two commands build an array **$props** that contains the names of all four profile path properties. Then the foreach statement loops through each property name one at a time, interpolating three subexpressions into a string:  
     **$p** # the loop variable, which holds the property name  
     **$PROFILE.$p** # the path stored in that property  
     **Test-Path $PROFILE.$p** # returns **True** if a file exists at that path, **False** if it doesn’t
  2. Enter those commands at *each* of your three PowerShell prompts. How many of these files already exist? Click or tap here to enter text.

# Task 2—Profile Script Content

Use a profile script for script code and aliases that add value to your interactive CLI work environment.

## Steps

1. Use Visual Studio Code to edit *your* profile on *your* computer:  
   **code $PROFILE**
   1. Add a **New-Alias** command to your profile script:  
        
      **New-Alias m more**
   2. Press [Ctrl]+S or tap **File** > **Save** to save your change. (If the file didn’t already exist, it will be created as soon as you save.)
2. Launch a new PowerShell instance, and try using the alias in place of more:  
   **Get-Help New-Alias | m**  
   Thanks to your new profile, from now on this new **m** alias will be available in your interactive PowerShell sessions. Press Q to quit **more** and return to the PowerShell prompt.
3. There’s a PSDrive for aliases. Using a splat **\*** wildcard, show all of the aliases that start with m:  
   **Get-ChildItem Alias:m\***   
   Find the row that shows your new **m** alias and record it here: Click or tap here to enter text.
4. *(Optional: not everybody likes having a single character alias for more. If you don’t want it, edit your profile again and change or remove it.)*

# Task 3—Create a function

Checking LAN IP addresses is a common troubleshooting task. When someone deploys a server but it can’t respond or communicate, among many things to diagnose are its IP address and subnet mask settings. To facilitate this, let’s write two functions:

* Get-IPNetwork: given an IPv4 address and a subnet mask as arguments, find and return the “network ID” (the IP address that represents the entire subnet).
* Test-IPNetwork: given two IPv4 addresses and a subnet mask, return **True** if both addresses are on the same subnet or **False** if they are on separate subnets.
  + Two IPv4 addresses are on the same subnet if and only if the network ID for each is the same. *This fact gives us an opportunity to re-use code: we’ll script* Get-IPNetwork *in this task, then we’ll use that function to implement* Test-IPNetwork *in the next task.*

Here’s some “pseudocode” (attempts at plain English instructions) for each function:

* Get-IPNetwork (parameters: IPv4 address, IPv4 subnet mask)  
  calculate the network ID by finding the *bitwise-and* of the address and subnet mask.  
  return: the network ID
* Test-IPNetwork (parameters: IPv4 address 1, IPv4 address 2, subnet mask)  
  to calculate network ID 1, call Get-IPNetwork with arguments (IPV4 address 1, subnet mask)  
  to calculate network ID 2, call Get-IPNetwork with arguments (IPV4 address 2, subnet mask)  
  compare results: if network ID 1 and network ID 2 are the same,  
   return True  
  otherwise  
   return False

## Steps

1. Using VSCode, create a new script file named Network.ps1. (For now, it’s okay to save the script in a temporary folder.)
2. Create the following code skeleton:  
   **function Get-IPNetwork {  
    param($IPAddr, $SubnetMask)  
    $SubnetID = $IPAddr -band $SubnetMask  
    return $SubnetID  
   }**
3. Run the script: [F5], or tap VSCode’s triangle  button.
   1. The script doesn’t produce any output. You should see something similar to this in VSCode’s terminal pane:  
      **PS C:\Users\alice> . 'C:\Temp\Network.ps1'  
      PS C:\Users\alice>**  
      The period **.** in front of the path to the script represents “dot-sourcing.” Dot-sourcing a script makes that script’s functions and variables available to subsequent commands in the terminal.
4. In the VSCode terminal, let’s try to use the new function.
   1. When you use a script function at a PowerShell prompt, the parameter names are the same as the **param** variable names in the script, but are prefixed with a hyphen **–** instead of a dollar **$**.
   2. Enter:   
      **Get-IPNetwork -IPAddr "192.168.3.4" -SubnetMask "255.255.255.0"**
      1. You should see an error message. Record it here: Click or tap here to enter text.
5. The *bitwise-and* operator **-band** only works on integer data types, not strings. Let’s convert the strings to objects of the IPAddress class. But before making changes to the script code, use the terminal in VSCode to experiment.
   1. Enter:   
      **[IPAddress] "192.168.3.4"**
      1. How many of the resulting object’s properties do you see in your output? Click or tap here to enter text.
   2. Let’s access and isolate useful properties encapsulated inside that object. Enter:   
      **([IPAddress] "192.168.3.4").IPAddressToString**
      1. What is the value of the **IPAddressToString** property? Click or tap here to enter text.
      2. What is that property’s type? Click or tap here to enter text.
      3. What is the value of the **Address** property? Click or tap here to enter text.
      4. What is the type of the **Address** property? Click or tap here to enter text.
   3. The subnet mask isn’t actually an IP address, but it has the identical structure, and for the purposes of the *bitwise-and* operation we are expected to treat it like one. Enter:   
      **[IPAddress] "255.255.255.0"**
      1. What is the value of its **IPAddressToString** property? Click or tap here to enter text.
      2. What is the value of its **Address** property? Click or tap here to enter text.
   4. As integer types, we can use bitwise-and on the Address properties. Enter:   
      **$i32\_Addr = ([IPAddress] "192.168.3.4").Address  
      $i32\_Mask = ([IPAddress] "255.255.255.0").Address  
      $i32\_Addr -band $i32\_Mask**
      1. Your output: Click or tap here to enter text.
      2. Now convert that integer result back into an IPAddress object. Enter:  
         **[IPAddress]** *<the number you recorded in step 5.4.1>*
      3. What’s the value of that object’s **IPAddressToString** property? Click or tap here to enter text. ← *That* is the network ID, 192.168.3.0. If your result differs, go back and fix!
   5. In these brief experiments, we “got a feel” for IPAddress objects, and we successfully got a correct subnet mask calculation.
6. Now go back to the VSCode editor pane and use what we learned to modify the function:   
      
   **function Get-IPNetwork {  
    param($IPAddr, $SubnetMask)  
    $i32\_Addr = ([IPAddress]$IPAddr).Address  
    $i32\_Mask = ([IPAddress]$SubnetMask).Address  
    $i32\_SubnetId = $i32\_Addr -band $i32\_Mask  
    $SubnetId = [IPAddress]$i32\_SubnetId  
    return $SubnetId.IPAddressToString  
   }**
   1. Run (“dot-source”) the updated script. Then try it again, with the subnet mask listed first:   
      **Get-IPNetwork -SubnetMask "255.255.255.0"-IPAddr "192.168.3.4"**
   2. Shortcut: as long as the arguments are *in the same order* as specified in the **param()** statement, you can omit the parameter names. We refer to this as using *positional parameters* instead of *named parameters*.  
      **Get-IPNetwork "192.168.3.4" "255.255.255.0"**

# Task 4—Reuse code by calling a function

Now that we can get a correct result from **Get-IPNetwork**, let’s use it to script **Test-IPNetwork**.

## Steps

1. After your Get-IPNetwork function, append the following function definition:  
   **function Test-IPNetwork {  
    param ($IPAddr1, $IPAddr2, $SubnetMask)  
    $SubnetId1 = Get-IPNetwork -IPAddr $IPAddr1 -SubnetMask $SubnetMask  
    $SubnetId2 = Get-IPNetwork -IPAddr $IPAddr2 -SubnetMask $SubnetMask  
    if ($SubnetId1 -eq $SubnetId2) {  
    return $true  
    }  
    else {  
    return $false  
    }  
   }**
   1. Run (“dot source”) the script again so that you can use this new function in the terminal pane.
2. Scenario: a server was deployed with the following settings:  
   IPv4 address 192.168.3.4  
   subnet mask 255.255.255.0  
   default gateway 192.168.3.1
   1. To learn whether the address and default gateway are on the same subnet, use **Test-IPNetwork** with positional parameters:  
      **Test-IPNetwork "192.168.3.4" "192.168.3.4" "255.255.255.0"**   
      Your output: Click or tap here to enter text.
3. Scenario: a server was deployed with these settings:  
   IPv4 address 10.10.49.121  
   subnet mask 255.255.248.0  
   default gateway 10.10.56.254
   1. For many people, this example is less obvious because of the unusual subnet mask.  
      **Test-IPNetwork "10.10.49.121" "10.10.56.254" "255.255.248.0"**   
      Your output: Click or tap here to enter text.
4. Scenario: the technician found that the default gateway was misconfigured. It should have been:  
   IPv4 address 10.10.49.121  
   subnet mask 255.255.248.0  
   default gateway 10.10.55.254
5. Enter:   
   **Test-IPNetwork "10.10.49.121" "10.10.55.254" "255.255.248.0"**   
   Your output: Click or tap here to enter text.
6. What happens when the argument strings don’t’ have IP addresses or subnet masks in them?  
   **Test-IPNetwork "server" "gw" "mask"**
   1. Your output or errors: Click or tap here to enter text.
   2. Okay, so the scripted functions so far aren’t perfect. We can “input” arguments into the functions that cause them to fail. (Later in the semester, you will have the opportunity to improve them, make them more robust, and provide enhanced functionality.)

# Task 5—Comment-based help documentation

Provide block comments that are parsed and displayed by the **Get-Help** cmdlet.

## Steps

1. Insert the following block comment inside the Get-IPNetwork function. Put it between the opening curly brace and the **param()** statement.  
    **<#  
    .SYNOPSIS  
    Given an IP address and a subnet mask, return its subnet ID  
    .DESCRIPTION  
    Given an IP address and a subnet mask, return the subnet ID  
    of the network on which that address resides.  
    .PARAMETER IPAddr  
    The IPv4 address as a "dotted-quad" string  
    .PARAMETER SubnetMask  
    The subnet mask as a "dotted-quad" string  
    .EXAMPLE  
    Get-IPNetwork -IPAddr "192.168.3.4" -SubnetMask "255.255.255.0"  
    returns "192.168.3.0"  
    #>**
   1. Run (“dot source”) the script again, then try using **Get-Help**:  
      **Get-Help Get-IPNetwork  
      Get-Help Get-IPNetwork -Examples  
      Get-Help Get-IPNetwork -Parameter IPAddr**
2. Write and test your own comment-based help for Test-IPNetwork. Record your block comment here: Click or tap here to enter text.

# Deliverable

Upload this document with completed answers to I-Learn Canvas.