Starting with Windows PowerShell

TABLE OF CONTENTS

PREFACE

This cheatsheet provides an overview of the most commonly used PowerShell commands, grouped by category. Whether you're new to PowerShell or an experienced user, this cheatsheet will serve as a handy reference guide for common tasks and commands.

From basic file management to advanced system administration, PowerShell can help you get things done more efficiently and effectively. This cheat sheet is designed to help you quickly find the right command for the job, so you can spend less time searching for information and more time getting things done.

We hope this cheatsheet helps you become more productive with PowerShell. Happy scripting!

INTRODUCTION

PowerShell is a command-line shell and scripting language developed by Microsoft for task automation and configuration management. It is built on the .NET Framework and offers a powerful set of cmdlets and tools for managing Windows systems and applications.

PowerShell has become the preferred tool for IT professionals who manage large-scale Windows environments, as it provides a standardized approach for managing systems and automating tasks.

PURPOSE AND BENEFITS OF POWERSHELL

PowerShell is designed to simplify and automate administrative tasks in Windows environments. It allows administrators to manage Windows systems more efficiently and effectively by providing a powerful, flexible, and scriptable command-line interface. Some benefits of using PowerShell include improved productivity, reduced errors, and increased scalability.

VERSIONS OF POWERSHELL

PowerShell has several versions, including PowerShell 1.0, 2.0, 3.0, 4.0, 5.0, 5.1, and 7.0, with the latest being PowerShell 7.1 at the time of writing.

SYSTEM REQUIREMENTS

The system requirements for PowerShell vary depending on the version and operating system being used. However, PowerShell 7.1 can run on Windows, macOS, and Linux systems.

INSTALLING POWERSHELL

PowerShell is installed by default on most modern Windows operating systems. However, if it's not installed, it can be downloaded and installed from the Microsoft website.

KEY FEATURES

- PowerShell is a powerful scripting language that allows administrators to automate tasks and create custom scripts.
- It provides a rich set of built-in cmdlets for performing common administrative tasks such as managing users, groups, and system settings.
- PowerShell supports remote administration of Windows systems, allowing administrators to manage multiple systems from a single console.
- PowerShell provides advanced error handling and debugging capabilities, making it easier to troubleshoot and resolve issues.
- PowerShell can be extended with third-party modules and tools, allowing administrators to customize and enhance its capabilities.

ARCHITECTURE

COMMAND LINE INTERFACE VS GRAPHICAL USER INTERFACE

The CLI interface provides users with greater flexibility, control, and automation capabilities, making it easier to perform repetitive administrative tasks. With PowerShell, administrators can automate tasks such as managing users and groups, configuring network settings, and managing virtual machines.

The GUI interface, on the other hand, provides a more intuitive and visual interface for managing systems. The GUI can be useful for tasks that require visual confirmation or interaction, such as configuring system settings, creating and configuring user accounts, and working with file

explorer.

However, for advanced administrative tasks, PowerShell's CLI is the preferred method, as it provides more control and automation capabilities. The GUI interface can be a helpful tool for basic system management tasks, but it can be time-consuming for complex and repetitive tasks.

CMDLETS

Cmdlets, or "command-lets," are the fundamental building blocks of PowerShell. Cmdlets are small, specialized commands that perform a specific task, such as retrieving information, modifying settings, or managing resources.

Here are some key things to know about cmdlets in PowerShell:

- Cmdlets are designed to be used in pipelines, allowing the output of one cmdlet to be used as input to another.
- Cmdlets follow a consistent naming convention, with a verb-noun structure. For example, `Get-Process` is a cmdlet that retrieves information about running processes, and `Set-Item` is a cmdlet that modifies the properties of a file or registry key.
- PowerShell includes a large number of built-in cmdlets, covering a wide range of system administration tasks.
- You can view a list of all available cmdlets using the `Get-Command` cmdlet.
- Cmdlets are designed to be easy to use and require minimal input from the user. They often include default values for parameters and support for aliases, allowing users to use familiar syntax and shortcuts.
- PowerShell also allows users to create custom cmdlets using PowerShell scripts or compiled code. This allows users to extend the functionality of PowerShell and create specialized cmdlets tailored to their specific needs.
- Cmdlets can be run on remote computers using PowerShell's remoting capabilities. This allows administrators to manage multiple systems from a central location, without needing to physically access each system.

Core cmdlets

These are the basic cmdlets that are included in PowerShell by default. They provide core functionality for working with files, directories, processes, and more.

Cmdlet	Description
Get-ChildItem	Lists files and folders in a directory
Set-Location	Changes the current directory
Get-Process	Lists running processes
Stop-Process	Stops a running process
Get-Service	Lists services on the system
Start-Service	Starts a stopped service
Stop-Service	Stops a running service
Get-Content	Displays the contents of a file
Set-Content	Sets the contents of a file
New-Item	Creates a new item, such as a file or folder
Remove-Item	Deletes an item

Active Directory cmdlets

These cmdlets are used for managing Active Directory objects, such as users, groups, and computers.

Cmdlet	Description
Get-ADUser	Lists user objects in Active Directory
New-ADUser	Creates a new user object
Set-ADUser	Modifies a user object
Remove-ADUser	Deletes a user object
Get-ADGroup	Lists group objects in Active Directory
New-ADGroup	Creates a new group object
Set-ADGroup	Modifies a group object
Remove-ADGroup	Deletes a group object

Cmdlet	Description
Add-ADGroupMember	Adds a member to a group
Remove-ADGroupMember	Removes a member from a group

Exchange Server cmdlets

These cmdlets are used for managing Exchange Server objects, such as mailboxes, contacts, and distribution groups.

Cmdlet	Description
Get-Mailbox	Lists mailboxes in Exchange Server
New-Mailbox	Creates a new mailbox
Set-Mailbox	Modifies a mailbox
Remove-Mailbox	Deletes a mailbox
Get-DistributionGroup	Lists distribution groups
New-DistributionGroup	Creates a new distribution group
Set-DistributionGroup	Modifies a distribution group
Remove-DistributionGroup	Deletes a distribution group
Add-DistributionGroupMember	Adds a member to a distribution group
Remove-DistributionGroupMember	Removes a member from a distribution group

SharePoint Server cmdlets

These cmdlets are used for managing SharePoint Server objects, such as sites, lists, and libraries.

Cmdlet	Description
Get-SPSite	Lists SharePoint sites
New-SPSite	Creates a new SharePoint site
Set-SPSite	Modifies a SharePoint site
Remove-SPSite	Deletes a SharePoint site
Get-SPWeb	Lists SharePoint webs
New-SPWeb	Creates a new SharePoint web
Set-SPWeb	Modifies a SharePoint web
Remove-SPWeb	Deletes a SharePoint web

SQL Server cmdlets

These cmdlets are used for managing SQL Server objects, such as databases, tables, and views.

Cmdlet	Description
Get-SqlDatabase	Lists SQL Server databases
New-SqlDatabase	Creates a new SQL Server database
Set-SqlDatabase	Modifies a SQL Server database
Remove-SqlDatabase	Deletes a SQL Server database
Get-SqlTable	Lists SQL Server tables
New-SqlTable	Creates a new SQL Server table
Set-SqlTable	Modifies a SQL Server table
Remove-SqlTable	Deletes a SQL Server table

Networking cmdlets

These cmdlets are used for working with network settings, such as IP addresses, DNS servers, and network adapters.

Cmdlet	Description
Test-Connection	Sends ICMP echo requests to a remote computer to test its availability.

Cmdlet	Description
Test-NetConnection	Tests the connection to a remote network resource by establishing a connection to a specified TCP port.
Get-NetAdapter	Retrieves a list of all network adapters installed on the computer.
Get-NetIPAddress	Retrieves IP address configuration information for all IP addresses assigned to network adapters on the computer.
Get-NetTCPConnection	Retrieves a list of all active TCP connections on the computer.
New-NetFirewallRule	Creates a new firewall rule for inbound or outbound traffic.

Security cmdlets

These cmdlets are used for working with security settings, such as user accounts, permissions, and certificates.

Cmdlet	Description
Get-Acl	Gets the access control list (ACL) for a resource.
Set-Acl	Changes the access control list (ACL) for a resource.
Get-PfxCertificate	Retrieves a certificate from a Personal Information Exchange (PFX) file.
New-SelfSignedCertificate	Creates a new self-signed certificate.
Export-Certificate	Exports a certificate from a certificate store to a file.
Import-Certificate	Imports a certificate from a file to a certificate store.

Cmdlet	Description
Test-NetConnection	Tests a network connection to a specified destination.
Set-ExecutionPolicy	Sets the script execution policy for the current user or computer.

Storage cmdlets

These cmdlets are used for working with storage devices and settings, such as disks, volumes, and file systems.

Cmdlet	Description
New-PSDrive	Creates a PowerShell drive that is connected to a network resource or a storage device.
Get-PSDrive	Gets the drives available in the current session.
Get-Volume	Retrieves information about the volumes on the system.
New-Item	Creates a new item (file, directory, registry key, etc.) at the specified location.
Remove-Item	Deletes the specified item.
Set-Item	Sets the value of a property of the specified item.
Get-Item	Gets the properties of the specified item.
Test-Path	Determines whether the specified path exists.
Rename-Item	Renames the specified item.
Get-ChildItem	Retrieves the items in one or more specified locations.
Copy-Item	Copies an item from one location to another.
Move-Item	Moves an item from one location to another.

Web cmdlets

These cmdlets are used for working with web-related technologies, such as HTTP requests, REST APIs, and HTML parsing.

Cmdlet	Description
Invoke-WebRequest	Sends an HTTP or HTTPS request to a web page and returns the response.
Invoke-RestMethod	Sends an HTTP or HTTPS request to a RESTful web service and returns the response.
ConvertTo-Json	Converts a PowerShell object to a JSON-formatted string.
ConvertFrom-Json	Converts a JSON-formatted string to a PowerShell object.
Test-NetConnection	Tests the availability of a network connection by performing a ping or a port test.
Resolve-DnsName	Resolves a DNS name to an IP address.

Note that these categories are not exhaustive and there may be some overlap between them. Also, not all categories may be relevant to your particular use case.

PROVIDERS

Providers in PowerShell are a way of accessing data stores in a hierarchical format, such as a file system, registry, or certificate store. Providers are similar to cmdlets in that they can be used in pipelines and follow a consistent naming convention, but they are specialized for working with specific data stores.

Here are some key things to know about providers in PowerShell:

- Providers are used to representing data stores in a way that is consistent with PowerShell's object-oriented nature. This allows data stores to be manipulated using PowerShell's built-in cmdlets and scripting capabilities.
- PowerShell includes several built-in providers, including the FileSystem provider for working with files and directories, the Registry provider for working with the Windows registry, and the Certificate provider for working with digital certificates.
- Providers are accessed using a PSDrive, which is a virtual drive that represents the data store. PSDrives are created using the `New-PSDrive` cmdlet and can be assigned a letter, a name, or any other unique identifier.
- Once a PSDrive is created, it can be accessed like a regular drive using PowerShell's built-in cmdlets and scripting capabilities. For example, the `Get-ChildItem` cmdlet can be used to retrieve a list of files and directories in the PSDrive.
- PowerShell also allows users to create custom providers using PowerShell scripts or compiled code. This allows users to extend the functionality of PowerShell and create specialized providers tailored to their specific needs.
- Providers can be used in pipelines along with cmdlets, allowing complex operations to be performed on data stores. For example, a script could retrieve a list of files using the FileSystem provider, filter the list using the `Where-Object` cmdlet, and then modify the files using the `Set-Item` cmdlet.

OBJECTS AND PIPELINES

In PowerShell, everything is an object. This means that all data types, including numbers, strings, and even commands, are represented as objects in PowerShell. Objects in PowerShell have properties, methods, and events, which can be manipulated and accessed using PowerShell's built-in cmdlets and scripting capabilities.

Here are some key things to know about objects and pipelines in PowerShell:

- Objects are created by cmdlets and returned as output to the pipeline. Each object has a set of properties, which can be retrieved using the `Select-Object` cmdlet or accessed directly using dot notation. For example, the `Get-Process` cmdlet returns a list of process objects, each

with properties such as `Name`, `Id`, and `CPU`.

- Pipelines in PowerShell are used to pass objects from one cmdlet to another, allowing complex operations to be performed on data without the need for intermediate variables. For example, the output of `Get-Process` can be passed to `Select-Object` to retrieve only the `Name` and `CPU` properties, and then to `Sort-Object` to sort the results by CPU usage.
- PowerShell's pipeline supports both filtering and sorting, as well as a variety of other operations such as grouping, joining, and formatting. This allows users to perform complex data manipulations using simple one-liners.
- PowerShell also supports the creation of custom objects using the `New-Object` cmdlet. This allows users to create objects with custom properties and methods, and then pass them to other cmdlets in the pipeline.
- PowerShell's object-oriented nature allows for easy integration with .NET Framework and other third-party libraries, allowing users to take advantage of existing code and APIs from within PowerShell scripts.
- One important thing to note is that objects in PowerShell are not always compatible with other command-line tools that expect text-based input and output. In such cases, PowerShell provides options for converting objects to text and vice versa, such as the `ConvertTo-Json` and `ConvertFrom-Json` cmdlets for working with JSON data.

MODULES

Modules in PowerShell are collections of cmdlets, functions, providers, and other resources that can be loaded and used in PowerShell. Modules allow users to extend the functionality of PowerShell beyond the built-in cmdlets and provide a way to share code and functionality with others.

Here are some key things to know about modules in PowerShell:

- PowerShell includes several built-in modules, such as the ActiveDirectory module for working with Active Directory, the Hyper-V module for managing virtual machines, and the NetTCPIP module for working with TCP/IP networking.
- Modules can be loaded using the `Import-Module` cmdlet or automatically loaded when needed based on module auto-discovery settings.
- Once a module is loaded, its cmdlets and functions can be used in the PowerShell session. For example, the `Get-ADUser` cmdlet in the ActiveDirectory module can be used to retrieve information about users in Active Directory.
- Modules can be managed using the `Get-Module`, `Import-Module`, and `Remove-Module` cmdlets, which allow users to list loaded modules, load new modules, and remove loaded modules, respectively.
- Users can create their own modules using PowerShell scripts and manifest files. This allows users to package their own cmdlets, functions, and other resources for easy distribution and use by others.
- Modules can also include other resources, such as scripts, configuration files, and help files, which can be accessed using the `Get-Command`, `Get-Help`, and `Get-Content` cmdlets, respectively.
- PowerShell also supports module versioning, which allows users to load and use specific versions of a module, ensuring compatibility with existing scripts and dependencies.
- Modules can be published to PowerShell galleries, such as the PowerShell Gallery or a private gallery, for easy sharing and installation by others.

VARIABLES

Variables in PowerShell are used to store and manipulate data in scripts and interactive sessions. PowerShell supports several different types of variables, each with its own scope and lifetime.

Here are some key things to know about variables in PowerShell:

- Variables in PowerShell are represented using a dollar sign `$` followed by the variable name. For example, `$myVariable` is a valid variable name in PowerShell.
- PowerShell supports several different types of variables, including scalar variables (which store a single value), array variables (which

store multiple values of the same type), and hash tables (which store key-value pairs).

- Variables in PowerShell have a scope, which determines where the variable is visible and accessible. PowerShell supports several different scopes, including global (which is visible throughout the entire session), local (which is visible only within the current script or function), and script (which is visible throughout the entire script but not outside of it).
- PowerShell also supports automatic variables, which are predefined variables that hold system information such as the current user, the last error message, and the current directory.
- PowerShell allows variables to be assigned values using the `=` operator. For example, `$myVariable = "Hello, World!"` assigns the string "Hello, World!" to the variable `$myVariable`.
- PowerShell also supports variable expansion, which allows variables to be used inside strings and other commands. Variable expansion is done using the `$()` syntax. For example, `"The value of my variable is $($myVariable)"` would expand to "The value of my variable is Hello, World!" if `$myVariable` was assigned the string "Hello, World!".
- PowerShell allows variables to be passed between cmdlets and functions using the pipeline. For example, `Get-ChildItem | Where-Object {$_.Name -like "*.txt"} | Select-Object FullName` retrieves a list of files, filters the list to include only files with a .txt extension, and then returns the full path of each file as a string object to the pipeline. These string objects can then be assigned to a variable for further processing or manipulation.

FUNCTIONS

Functions in PowerShell are reusable blocks of code that perform a specific task. They allow you to write complex scripts more easily by breaking them down into smaller, more manageable pieces.

Here are some key things to know about functions in PowerShell:

- Functions in PowerShell are defined using the `function` keyword, followed by the function name, any parameters the function accepts, and the body of the function enclosed in braces. For example, `function MyFunction { Write-Output "Hello, World!" }` defines a function called `MyFunction` that simply writes the string "Hello, World!" to the console.
- Functions in PowerShell can accept parameters, which are used to pass data into the function. Parameters are defined inside the parentheses following the function name, separated by commas. For example, `function MyFunction ($Name) { Write-Output "Hello, $Name!" }` defines a function called `MyFunction` that accepts a parameter called `$Name` and writes a personalized greeting to the console.
- Functions in PowerShell can return values using the `return` keyword. For example, `function Add-Numbers ($a, $b) { return $a + $b }` defines a function called `Add-Numbers` that accepts two parameters, adds them together, and returns the result.
- Functions in PowerShell can be saved to disk as script files, just like regular PowerShell scripts. This allows you to reuse functions across different scripts and sessions.
- PowerShell supports several different types of functions, including advanced functions (which provide additional features like parameter validation and pipeline input), script functions (which allow you to define a function as a separate script file), and anonymous functions (which are defined inline as part of a larger command).
- PowerShell functions can be called from other functions, scripts, or interactive sessions, making them a powerful tool for building complex automation workflows.

SCRIPTS

Scripts in PowerShell are text files that contain a series of PowerShell commands and statements. They allow you to automate tasks by running the same set of commands multiple times, or by running them on a schedule.

Here are some key things to know about scripts in PowerShell:

- PowerShell scripts are saved as plain text files

with a `.ps1` file extension. They can be edited in any text editor, including the built-in PowerShell Integrated Scripting Environment (ISE).

- PowerShell scripts are executed by invoking the `powershell.exe` executable and passing the script file as a command-line argument. For example, `powershell.exe -File C:ScriptsMyScript.ps1` runs a script file called `MyScript.ps1` located in the `C:Scripts` directory.
- PowerShell scripts can contain any valid PowerShell command or statement, including cmdlets, functions, variables, loops, and conditional statements. They can also include comments, which are denoted by the `#` symbol.
- PowerShell scripts can accept parameters, which are used to pass data into the script. Parameters are defined using the `param` keyword at the beginning of the script, followed by the parameter names enclosed in parentheses. For example, `param($Name) Write-Output "Hello, $Name!"` defines a script that accepts a parameter called `$Name` and writes a personalized greeting to the console.
- PowerShell scripts can be run on a schedule using the Windows Task Scheduler. This allows you to automate tasks like backups, report generation, and system maintenance without manual intervention.
- PowerShell scripts can be used to build more complex automation workflows by calling other scripts, functions, or cmdlets. This allows you to break down large tasks into smaller, more manageable pieces that can be reused across different scripts and sessions.

REMOTING

PowerShell Remoting allows you to execute PowerShell commands and scripts on remote computers. It enables you to manage large numbers of computers or servers from a single, centralized location, without having to physically access each one.

Here are some key things to know about PowerShell Remoting:

- PowerShell Remoting relies on the Windows Remote Management (WinRM) service to establish a connection between the local computer and the remote computer. WinRM must be enabled and configured on both the local and remote computers before PowerShell Remoting can be used.
- PowerShell Remoting can be initiated from the local computer using the `Enter-PSSession` cmdlet or the `Invoke-Command` cmdlet. `Enter-PSSession` creates an interactive session with the remote computer, allowing you to enter commands as if you were physically present at the remote computer. `Invoke-Command` executes a single command or script on the remote computer and returns the results to the local computer.
- PowerShell Remoting requires that you have administrative privileges on both the local and remote computers in order to establish a connection and execute commands. You must also have the appropriate permissions to access and modify the resources on the remote computer.
- PowerShell Remoting supports several authentication methods, including Kerberos, Negotiate, and Basic authentication. You can also use Secure Sockets Layer (SSL) or Transport Layer Security (TLS) to encrypt the connection between the local and remote computers.
- PowerShell Remoting allows you to manage multiple remote computers simultaneously using a single command. This makes it easy to perform common administrative tasks like software installation, patching, and configuration changes across a large number of computers.
- PowerShell Remoting can be configured to run commands in the background, allowing you to continue working on other tasks while the remote command executes. This is especially useful for long-running tasks like software installations or system backups.
- PowerShell Remoting can be used in combination with other PowerShell features like workflows and Desired State Configuration (DSC) to build complex automation workflows and ensure consistent configuration across large numbers of computers.

THE LANGUAGE

SYNTAX AND GRAMMAR

PowerShell has its own syntax and grammar rules that govern how commands and scripts are written and executed. Here are some key things to know about the syntax and grammar of PowerShell:

- PowerShell commands are composed of cmdlets, parameters, and arguments. A cmdlet is a PowerShell command that performs a specific action, like `Get-Process` or `Set-Item`. Parameters modify the behavior of a cmdlet and are specified using a dash followed by the parameter name, like -Name or -Path. Arguments are values that are passed to a cmdlet or parameter and are usually enclosed in parentheses or quotation marks, like `(Get-ChildItem)` or `"C:MyFolder"`.
- PowerShell uses a pipeline to pass objects between cmdlets. A pipeline is denoted by the `|` symbol and allows you to chain together multiple cmdlets to perform complex operations. For example, `Get-ChildItem | Where-Object {$_.Name -like "*.txt"}` retrieves all files in the current directory with an `.txt` extension.
- PowerShell uses a set of reserved keywords to define its grammar and syntax. These include keywords like `if`, `else`, `for`, and `while`, which are used to create conditional statements and loops in PowerShell scripts.
- PowerShell uses a variable naming convention that begins with a `$` symbol, followed by a name that describes the purpose of the variable. For example, `$Name` might be used to store the name of a user or computer.
- PowerShell scripts are saved with a `.ps1` file extension and can be executed from the PowerShell console or from the command line using the `powershell.exe` executable. PowerShell scripts can also be run on a schedule using the Windows Task Scheduler.
- PowerShell scripts can be debugged using a built-in debugging environment that allows you to step through each line of code and view the values of variables and expressions. Debugging can be initiated by running a script with the `-Debug` parameter, or by setting breakpoints within the script using the `Set-PSBreakpoint` cmdlet.
- PowerShell scripts can be written using a variety of text editors and integrated development environments (IDEs), including the built-in PowerShell Integrated Scripting Environment (ISE), Visual Studio Code, and Notepad++. These tools provide features like syntax highlighting, code completion, and debugging support to make it easier to write and test PowerShell scripts.

DATA TYPES

PowerShell supports a variety of data types that can be used to store and manipulate values in scripts and commands. Here are some of the most common data types in PowerShell:

Types	Description
Strings	A string is a sequence of characters that is enclosed in quotation marks. Strings can be concatenated using the `+` operator, or interpolated into other strings using the `$` symbol. For example, `"Hello, " + "world"` would produce the string "Hello, world"
Integers	An integer is a whole number that can be positive or negative. Integers can be used in arithmetic operations like addition, subtraction, multiplication, and division. For example, `$x = 10 + 5` would assign the value 15 to the variable `$x`

Types	Description
Booleans	A boolean is a value that is either true or false. Booleans can be used in conditional statements and loops to control program flow. For example, `if ($x -eq 10) { "x is equal to 10" }` would output the message "x is equal to 10" if the variable `$x` has the value 10
Arrays	An array is a collection of values that can be accessed using an index. Arrays can be created using the `@()` operator, and elements can be added or removed using methods like `Add()` and `Remove()`. For example, `$myArray = @(1, 2, 3)` would create an array with three elements
Hashtables	A hashtable is a collection of key-value pairs that can be accessed using the key. Hashtables can be created using the `@{}` operator, and values can be added or removed using methods like `Add()` and `Remove()`. For example, `$myHash = @{ Name = "John"; Age = 30 }` would create a hashtable with two keys, "Name" and "Age"
Objects	An object is a complex data type that contains properties and methods. Objects can be created using the `New-Object` cmdlet, and properties can be accessed using dot notation. For example, `$myObject = New-Object -TypeName PSObject -Property @{ Name = "John"; Age = 30 }` would create an object with two properties, "Name" and "Age". The value of the "Name" property could be accessed using `$myObject.Name`

VARIABLES AND OPERATORS

Variables and operators are essential components of any programming language, including PowerShell. Here is a brief overview of variables and operators in PowerShell:

Variables

In PowerShell, variables are used to store data that can be used and manipulated in scripts and commands.

Variables are defined using the `$` symbol followed by the variable name. For example:

```
$myVariable = "Hello, world!"
```

would assign the string "Hello, world!" to the variable `$myVariable`.

Variables can be used in commands and expressions by enclosing them in curly braces `{}`. For example:

```
Write-Host "The value of my variable is: $($myVariable)"
```

would output the message "The value of my variable is: Hello, world!".

Operators

Operators are used to perform mathematical and logical operations on values and variables.

Operators	Description
+	addition
-	subtraction
*	multiplication
/	division
%	modulus
-eq	equal to
-ne	not equal to
-lt	less than
-gt	greater than
-le	less than or equal to
-ge	greater than or equal to
-and	logical and
-or	logical or
-not	logical not

Examples:

- `$x = 10 + 5` assigns the value 15 to the variable `$x`.
- `if ($x -eq 15) { Write-Host "x is equal to 15" }` would output the message "x is equal to 15" if the variable `$x` has the value 15.
- `$y = $x * 2` assigns the value 30 to the variable `$y`.
- `if ($x -gt 5 -and $y -lt 40) { Write-Host "Both conditions are true" }` would output the message "Both conditions are true" if the variable `$x` has a value greater than 5 and the variable `$y` has a value less than 40.

Note that PowerShell also supports a variety of other operators and types, such as bitwise operators, regular expression operators, and arrays. The examples above are just a small sampling of what is possible with variables and operators in PowerShell.

Bitwise Operators

In PowerShell, bitwise operators are used to manipulate binary values at the bit level. Here are the bitwise operators available in PowerShell:

Operator	Description
-bnot	Flips all bits in the input value
-bor	Sets each bit in the result to 1 if either or both corresponding bits in the input values are 1
-bxor	Sets each bit in the result to 1 if only one of the corresponding bits in the input values is 1
-band	Sets each bit in the result to 1 if both corresponding bits in the input values are 1
-shl	Shifts the bits of the input value to the left by the specified number of positions
-shr	Shifts the bits of the input value to the right by the specified number of positions

Here's an example of using bitwise operators in PowerShell:

```
$a = 0b0110
$b = 0b1010

# Bitwise NOT
$c = -bnot $a
Write-Host "Bitwise NOT:
$($c.ToString('X'))" # Output: FFFF
FFF9

# Bitwise OR
$c = $a -bor $b
Write-Host "Bitwise OR:
$($c.ToString('X'))" # Output: 1E

# Bitwise XOR
```

```
$c = $a -bxor $b
Write-Host "Bitwise XOR:
$($c.ToString('X'))" # Output: 14

# Bitwise AND
$c = $a -band $b
Write-Host "Bitwise AND:
$($c.ToString('X'))" # Output: 2

# Shift left
$c = $a -shl 1
Write-Host "Shift left:
$($c.ToString('X'))" # Output: C

# Shift right
$c = $b -shr 1
Write-Host "Shift right:
$($c.ToString('X'))" # Output: 5
```

WILDCARDS

In PowerShell, wildcards are characters that allow you to perform pattern matching when working with strings or paths. They are used with various cmdlets and operators to search, filter, and replace text.

Here are some commonly used wildcards in PowerShell:

Wildcard	Description
*	Matches zero or more characters in a string or path. For example, `Get-ChildItem C:Windows*` returns all files and folders in the `C:Windows` directory
?	Matches any single character in a string or path. For example, `Get-ChildItem C:WindowsSystem32?calc.exe` returns all versions of the `calc.exe` file in the `C:WindowsSystem32` directory
[ ]	Matches any single character within the specified range or set. For example, `Get-ChildItem C:WindowsSystem32[abc]*` returns all files and folders in the `C:WindowsSystem32` directory that begin with the letters "a", "b", or "c"
-	Specifies a range of characters within `[ ]`. For example, `Get-ChildItem C:WindowsSystem32[a-z]*` returns all files and folders in the `C:WindowsSystem32` directory that begin with any lowercase letter
{ }	Specifies a set of alternative characters. For example, `Get-ChildItem C:Windows{.exe,.dll}` returns all files in the `C:Windows` directory that have either a `.exe` or `.dll` extension

Here are some cmdlets and operators that support wildcards:

Cmdlet	Description
Get-ChildItem	Allows you to search for files and folders using wildcards
Select-String	Allows you to search for text in a file or string using wildcards
-like -notlike	Allows you to filter strings using wildcards
-match -notmatch	Allows you to search for patterns in a string using regular expressions

Here are some examples of using wildcards in PowerShell:

```
# Find all files with "log" in their
name in the C:WindowsLogs directory
Get-ChildItem C:WindowsLogs*log*

# Find all files with a ".txt"
extension in the C:UsersPublic
directory
Get-ChildItem C:UsersPublic*.txt

# Search for the word "error" in all
files with a ".log" extension in the
C:WindowsLogs directory
Get-ChildItem C:WindowsLogs*.log |
Select-String "error"

# Filter all processes whose name
begins with "w" using the -like
operator
Get-Process | Where-Object { $_.Name
-like "w*" }

# Search for all strings in an array
that match a pattern using the
-match operator
$strings = "apple", "banana",
"cherry"
$strings -match "a"
```

CONDITIONAL STATEMENTS

Conditional statements are an important part of any programming language, and PowerShell is no exception.

If-Else statements

If-Else statements allow you to execute different blocks of code based on whether a certain condition is true or false.

The basic syntax for an If-Else statement is as follows:

```
if (condition) {
    # Code to execute if condition
is true
} else {
    # Code to execute if condition
is false
}
```

The `if` keyword is followed by the condition to check in parentheses, and the code to execute if the condition is true is enclosed in curly braces `{}`.

If the condition is false, the code to execute in the `else` block is executed instead.

You can also chain multiple conditions together using the `elseif` keyword:

```
if (condition1) {
    # Code to execute if condition1
is true
} elseif (condition2) {
    # Code to execute if condition2
is true
} else {
    # Code to execute if neither
condition1 nor condition2 is true
}
```

You can also use the `-and` and `-or` operators to combine conditions together:

```
if (condition1 -and condition2) {
    # Code to execute if both
condition1 and condition2 are true
}

if (condition1 -or condition2) {
    # Code to execute if either
condition1 or condition2 is true
}
```

Ternary Operator

In PowerShell, the ternary operator allows you to perform a simple comparison and execute one of two expressions based on the result.

Here's an example of using the ternary operator in PowerShell:

```
$number = 10

$result = if ($number -gt 5) {
"Greater than 5." } else { "Less
than or equal to 5." }

# Using ternary operator instead
$result = $number -gt 5 ? "Greater
than 5." : "Less than or equal to
5."

Write-Host $result
```

Note that the ternary operator is useful for simple comparisons and expressions, but can quickly become unwieldy for more complex scenarios. In those cases, it's better to use an `if` statement for clarity and readability.

Switch

In PowerShell, `switch` statements are used to compare a single value against a set of possible values and perform different actions based on the match. Here is the basic syntax for a `switch` statement:

```
switch -exact ($value) {
    <value1> { <action1> }
    <value2> { <action2> }
    <value3> { <action3> }
    default { <default-action> }
}
```

`Switch` statements can also be used with regular expressions or wildcards to match against patterns. To use a regular expression, use the `-regex` parameter instead of `-exact`. To use a wildcard pattern, use the `-wildcard` parameter instead of `-exact`.

```
$name = "John Doe"

switch -wildcard ($name) {
    "*Doe" { Write-Host "The name
ends with Doe." }
    "John*" { Write-Host "The name
starts with John." }
    "*D*" { Write-Host "The name
contains the letter D." }
    default { Write-Host "No match
found." }
}
```

LOOPS

Loops are an essential part of any programming language, and PowerShell provides several types of loops for repetitive operations. Here is an overview of the For, ForEach, and While loops in PowerShell:

For Loop

The For loop is a classic loop that allows you to iterate over a range of values a specific number of times.

The basic syntax for a For loop is as follows:

```
for ($i = 0; $i -lt 10; $i++) {
    # Code to execute for each
iteration
}
```

The first statement initializes a counter variable `$i` to 0, the second statement checks if the counter is less than 10, and the third statement increments the counter by 1 after each iteration.

The code to execute for each iteration is enclosed in curly braces `{}`.

ForEach Loop

The ForEach loop allows you to iterate over a collection of items, such as an array or a list.

The basic syntax for a ForEach loop is as follows:

```
foreach ($item in $collection) {
    # Code to execute for each item
in the collection
}
```

The `$item` variable is assigned each item in the

collection in turn, and the code to execute for each item is enclosed in curly braces `{}`.

While Loop

The While loop allows you to repeatedly execute a block of code while a certain condition is true.

The basic syntax for a While loop is as follows:

```
while (condition) {
    # Code to execute while
condition is true
}
```

The code to execute is enclosed in curly braces `{}`, and the condition is checked at the beginning of each iteration.

Examples:

- `for ($i = 0; $i -lt 10; $i++) { Write-Host $i }` would output the numbers 0 through 9, one per line.
- `$colors = @("red", "green", "blue") ; foreach ($color in $colors) { Write-Host $color }` would output the strings "red", "green", and "blue", one per line.
- `$i = 0 ; while ($i -lt 10) { Write-Host $i ; $i++ }` would output the numbers 0 through 9, one per line.

DATA STRUCTURES

Arrays and Hashtables are two important data structures in PowerShell that allow you to store and manipulate collections of values. Here is an overview of Arrays and Hashtables in PowerShell:

Arrays

An array is a collection of items of the same data type, such as strings or integers.

In PowerShell, you can create an array by enclosing a comma-separated list of values in parentheses `()`.

You can access individual items in an array by their index, which starts at 0. For example:

`$array[0]`

would return the first item in the array.

You can add items to an array using the `+=` operator. For example:

`$array += "new item"`

would add the string "new item" to the end of the array.

Example:

```
# Create an array of strings
$colors = ("red", "green", "blue")

# Access individual items in the
array
Write-Host $colors[0]   # Output:
red

# Add a new item to the end of the
array
$colors += "yellow"
Write-Host $colors      # Output:
red green blue yellow
```

Hashtables

A hashtable is a collection of key-value pairs, where each key is unique and maps to a specific value.

In PowerShell, you can create a hashtable using the `@{}` notation, with each key-value pair separated by a semicolon `;`.

You can access the value for a specific key using the `$hashtable[key]` notation. For example:

`$hashtable["key1"]`

would return the value associated with the key "key1".

You can add a new key-value pair to a hashtable using the `$hashtable[key] = value` notation.

Example:

```
# Create a hashtable of key-value
pairs
$ages = @{
```

```
    "John" = 30
    "Jane" = 25
}

# Access the value for a specific
key
Write-Host $ages["John"]    #
Output: 30

# Add a new key-value pair
$ages["Bob"] = 40
Write-Host $ages            #
Output: {John=30; Jane=25; Bob=40}
```

Note that PowerShell also supports other types of collections, such as Lists and Queues, which may be useful in specific situations. However, Arrays and Hashtables are the most commonly used data structures in PowerShell scripts.

Lists

A list is a collection of items that can be of any type. In PowerShell, you can create a list using the `@()` notation. Here's an example:

```
$list = @(1, 2, 3, "four", "five")
```

Adding Elements to a List

You can add elements to a list using the `+=` operator. Here's an example:

```
$list = @()
$list += 1
$list += 2, 3, 4
$list += "five"
```

Accessing Elements of a List

You can access elements of a list using the `[]` operator. Here's an example:

```
$list = @(1, 2, 3, "four", "five")
$list[0]     # returns the element
at index 0
$list[-1]    # returns the last
element of the list
```

Slicing an Array

You can extract a slice of an array using the `..` operator. Here's an example:

```
$list = @(1, 2, 3, "four", "five")
$slice = $list[1..3]    # returns
the elements at index 1, 2, and 3
```

Removing Elements from a List

You can remove elements from a list using the `Remove()` method. Here's an example:

```
$list = @(1, 2, 3, "four", "five")
$list.Remove(2)    # removes the
element at index 2
```

REGULAR EXPRESSIONS

Regular Expressions (regex or regexp) is a powerful tool used for pattern matching and text manipulation. In PowerShell, you can use regular expressions with a variety of cmdlets and operators to perform advanced text processing tasks. Here's an overview of how to use regular expressions in PowerShell:

`Select-String` : Is used to search for patterns in strings or files using regular expressions. It returns the matching lines and the patterns found in those lines.

```
# Search for the pattern "error" in
a file using regular expressions
Get-Content C:log.txt | Select-
String -Pattern "error"
```

`-match` : Is used to check if a string matches a regular expression pattern. It returns a Boolean value.

```
# Check if a string matches a
regular expression pattern
```

```
if ($string -match "pattern") {
    # do something
}
```

`-replace` : Is used to replace text that matches a regular expression pattern with a new string.

```
# Replace text that matches a
regular expression pattern
$string -replace "pattern", "new
string"
```

Regular expression patterns can include special characters and metacharacters to match specific types of characters or patterns. For example, the `.` metacharacter matches any single character, while `*` matches zero or more occurrences of the preceding character or group.

```
# Use a regular expression pattern
to match an IP address
if ($string -match
"d{1,3}.d{1,3}.d{1,3}.d{1,3}") {
    # do something
}
```

PowerShell also includes a number of regular expression operators and flags that can be used to modify how regular expressions are matched. For example, the `-regex` operator can be used to perform case-insensitive matching, while the `-split` operator can be used to split a string into an array using a regular expression pattern as the delimiter.

```
# Perform a case-insensitive match
using regular expressions
if ($string -iregex "pattern") {
    # do something
}

# Split a string into an array using
a regular expression pattern as the
delimiter
$array = $string -split "s+"
```

ERROR HANDLING

In PowerShell, you can use the `Try-Catch` statement to handle errors that might occur during script execution. The `Try` block contains the code that might throw an error, while the `Catch` block contains the code that is executed if an error occurs.

Here's an example of using `Try-Catch` statement in PowerShell:

```
try {
    # Code that might throw an error
    Get-ChildItem -Path
"C:SomePathThatDoesNotExist"
}
catch {
    # Code that handles the error
    Write-Host "An error occurred:
$($_.Exception.Message)"
}
```

You can also use the `Finally` block to specify code that will always be executed, regardless of whether an error occurs. This can be useful for tasks such as cleaning up resources or logging errors.

Here's an example of using `Try-Catch-Finally` statement in PowerShell:

```
try {
    # Code that might throw an error
    $file = Get-Content
"C:SomeFileThatDoesNotExist"
}
catch {
    # Code that handles the error
    Write-Host "An error occurred:
$($_.Exception.Message)"
}
finally {
    # Code that is always executed
    Remove-Item -Path
"C:SomeTempFile.txt" -Force
}
```

DEBUGGING

In PowerShell, you can use several methods to debug your scripts, including:

- **Debugging with the PowerShell ISE** : If you are using the PowerShell ISE, you can use the built-in debugging features to step through your code, set breakpoints, and inspect variables. To enable debugging, click on the Debug menu, and select Start Debugging or press F5.
- **Debugging with Visual Studio Code** : If you are using Visual Studio Code, you can use the PowerShell extension to debug your scripts. To enable debugging, add a breakpoint to your code by clicking on the line number, then press F5 to start debugging.
- **Adding Write-Debug statements** : You can add `Write-Debug` statements to your code to output debug information. These statements will only be displayed if you run your script with the `-Debug` parameter.
- **Using the Set-PSDebug cmdlet** : You can use the `Set-PSDebug` cmdlet to enable or disable debug mode. When debug mode is enabled, PowerShell will display additional debug information, such as the current line of code and the values of variables.

Here's an example of using `Write-Debug` statements to debug a script:

```
function Test-Debug {
    Write-Debug "Starting function"
    $var1 = "Hello"
    Write-Debug "Var1 value: $var1"
    $var2 = "World"
    Write-Debug "Var2 value: $var2"
    $result = "$var1 $var2"
    Write-Debug "Result value:
$result"
    return $result
}

Write-Debug "Starting script"
$result = Test-Debug
Write-Host "Result: $result"
```

The `Starting script` statement is also a `Write-Debug` statement, but since debug mode is not enabled, it will not be displayed. To run the script in debug mode, run it with the `-Debug` parameter:

```
.MyScript.ps1 -Debug
```

When run in debug mode, the script will output the `Write-Debug` statements, as well as additional debug information.