Powershell Fundamentals

#Basic Vocabulary for Powershell
PS uses commands called Cmdlets (Commandlets)
Cmdlets are built on a common runtime vs an executable.
All inputs and outputs are .NET objects.

#Cmdlets
Take an object input and return an object.
Cmdlets are built on the .NET core and are open source.
It is possible to build your cmdlet in a .NET core or PS.

#Parameters and Arguments
Cmdlets support the use of parameters as part of the input mechanism.
Parameters can be added to the cmdlet at the command line or passed to cmdlets through the pipeline.

Arguments
The values of each parameter detail the actual input that the cmdlet will accept, how it should work, and what data it outputs, if any.

#Check a cmdlet using the Powershell Kata
Syntax
<Cmdlet> | where | where-object | ? {$_.<properties of the object you want to search> "String"}

#How do you search for what properties you can use
You can find out what properties are available on the objects returned by `Get-Service` using the following methods.
The `Get-Member` cmdlet (`gm` for short) shows all properties and methods of an object:

Syntax
<cmdlet> | Get-Member -MemberType Property
This will output a list of properties, such as:

```
Name        MemberType   Definition
----        ----------   ----------
Name        Property     string Name {get;}
DisplayName Property     string DisplayName {get;}
Status      Property     System.ServiceProcess.ServiceControllerStatus Status {get;}
StartType   Property     System.ServiceProcess.ServiceStartMode StartType {get;}
```

Powershell Fundamentals

Getting Help in Powershell

#How to use Get-Help
Use the command
get-help -name <cmdlet>
To get an example of a cmdlet
get-help -name <cmdlet> -examples
To search for keywords in the get-help command
Get-help -name <cmdlet> -ShowWindow
How to search for a string of information for a cmdlet in Powershell
Get-help "Brief synopsis of the cmdlet that you are looking for"

he get-help command will display the following information without the - full operation.

- **NAME**
- **SYNOPSIS**
- **SYNTAX**
- **DESCRIPTION**
- **RELATED LINKS**
- **REMARKS**

#How to use the Get-Command
The Get-Command cmdlet retrieves all commands installed on the computer, including cmdlets, aliases, functions, filters, scripts, and applications.

Syntax
get-Command
Will print a list of all cmdlets, aliases, functions, filters, scripts, and applications.
Get-Command <cmdlet>
It will display the information on a specific cmdlet.
Get-Commadn -verb <verb> or Get-Command -noun <noun>
Will search for the Verb portion of a cmdlet or the noun portion with <-noun>
Get-Command -Name *string of text to search* -CommandType <cmdlet, alisas, or function>
Search for a cmdlet, function, or alias based only on the text in the wildcard *___*

#Wildcard for filtering
A wildcard or the * symbol simply matches any and all string filters.
Ex.
Get-Command Get*
This will call the Get-Command to look for any command that starts with Get.

Powershell Fundamentals

#How to use the Get-Member cmdlet
The Get-Member cmdlet gets the members, properties, and methods of objects.

Syntax
<cmdlet> | Get-Member
Now we can see all the methods and properties of that object. Think of the methods as "things we can do with these objects," and properties as "things these objects are."

We can filter the Get-Member command further by adding the following.

<cmdlet> | Get-Member -MemberType <Method, AliasProperty, Event, Property, ect.>

```
PS C:\WINDOWS\system32> Get-Process| Get-Member -MemberType Method


   TypeName: System.Diagnostics.Process

Name                      MemberType Definition
----                      ---------- ----------
BeginErrorReadLine        Method     void BeginErrorReadLine()
BeginOutputReadLine       Method     void BeginOutputReadLine()
CancelErrorRead           Method     void CancelErrorRead()
CancelOutputRead          Method     void CancelOutputRead()
Close                     Method     void Close()
CloseMainWindow           Method     bool CloseMainWindow()
CreateObjRef              Method     System.Runtime.Remoting.ObjRef CreateObjRef(type requestedType)
Dispose                   Method     void Dispose(), void IDisposable.Dispose()
Equals                    Method     bool Equals(System.Object obj)
GetHashCode               Method     int GetHashCode()
GetLifetimeService        Method     System.Object GetLifetimeService()
GetType                   Method     type GetType()
InitializeLifetimeService Method     System.Object InitializeLifetimeService()
Kill                      Method     void Kill()
Refresh                   Method     void Refresh()
Start                     Method     bool Start()
ToString                  Method     string ToString()
```

Powershell Alias
An alias is an alternate name for a cmdlet, function, or executable file, including scripts. PowerShell includes a set of built-in aliases.

#How to create an Alias
Syntax
Set-Alias -Name <Alias Name> -Value <cmdlet, function, .exe, etc. that we want an alias for>

Ex. Set-Alias -Name List -Value Get-Childitem
You can use the Get-Alias cmdlet to verify that the alias you created worked.

Get-Alias -Name <Alias name>
Ex. Get-Alias -Name List.

Powershell Fundamentals

To get help with the get-alias command
Get-help get-alias

How to get the alias of a cmdlet
Get-alias <cmdlet name>

How do I get all the aliases for a cmdlet
Get-alias -Definition <cmdlet name>

How to set a new alias for a cmdlet
Set-alias -name <alias name> -value <cmdlet that you want to add an alias to>

Note! Do-alias: Only really used when making a brand new alias

Variables

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

There are Three Types of Variables
User Created: created and maintained by the user.
Automatic: Created by PowerShell, PowerShell changes its values to maintain accuracy.
Preference: Store user preferences for PowerShell. These variables are created by PowerShell and are populated with default values. Users can change the values of these variables.

#User Created Variables
Variables you create at the PowerShell command line exist only while the PowerShell window is open. To save a User Created Variable it has to be added to your Powershell Profile.

Syntax for a Variable
$<Variable name> = <Value>

#How to save a User Created Variable to your profile.

1. **First, run the following command. Test-Path $profile**
2. **The output should print FALSE**
3. **Type New-Item -Path $profile -Type File -Force**
4. **Re-run the Test-Path $profile**
5. **Output should return TRUE**

Powershell Fundamentals

Arrays and Hash Tables

#How to create a Array in Powershell and call the array
Syntax Notes:
Any integers are separated by a comma and anything that you want classified as a string must be enclosed in quotation marks.

$<Array Name> = <Values or stings that you want in the array>

Example.
$A = 1, 2, 3, 4, 5 “Numbers”

Echo $A
If done properly the output should be “1,2,3,4,5 Numbers”

The data in our example Array is now in associated memory called an “index”. With the array name of $A. When we want to access this data we can interact with the array and index its stored. The chart below explains how the array of integers and strings is stored in memory.

Powershell always starts an array value from an index of 0.

Int or string	Memory Location
1	0
2	1
3	2
4	3
5	4
Numbers	5

Knowing where the integers or strings are stored in the array we can now check the index for that value.
Ex.
Syntax
Echo $<Array Name>[Index]
Echo $A[0]
1
If the Array is extremely large we can use negative numbers to reference the last value of the array as seen below.

Powershell Fundamentals

Integer or String	Memory Location	Negative Value
1	0	-5
2	1	-4
3	2	-3
4	3	-2
5	4	-1

If we use the same process to echo an index of an integer or string but instead use a [-1] to tell Powershell that we want to reference the last value of the array. Or the second to last with [-2] and so on and so forth.

```
Syntax
Echo $A[-1]
5
```

```
#Modifying an Array
```

To add an integer or string to an existing Array you can use the "+=" operator.

Syntax for adding to a Array

```
$<Array Name>  +=  <single value for a integer or " " for a string>
```

$A += "Apples"

```
#How to make a Loop in a Array

$numbers = 1, 2, 3, 4, 5

foreach ($num in $numbers) {
        Write-Output "Number: $num"
}
```

Hash Tables

"Array Subexpression"

A Hash Table is a directory or an associative array. It is a compact data structure that stores one or more key/value pairs. Hash tables can contain information like a machine's name and IP address.

Powershell Fundamentals

Hash tables use a Key and Value System. Where the value is associated with a Key. Unlike Arrays Hash Tables will use Key based indexing. You cannot reference the value's numerical position but instead reference its Key.

#How to create a Hash Table
Syntax
$<Hash Table Name> = @{ Key = “Value”, Key = “Value”, Key = “Value”}

Ex.
$Food = @{ A = “Apple”; B = “ Banana”; C = “Candy”}

#How to Reference a Key in a Hash Table
Syntax
Echo $<hash table>[“Key”]

Ex.
Echo $Food[“A”]
Apple

A Hash Table follows the same kind of rules such as an Excel Spreadsheet table.

Key A	Key B	Key C
Value Apple	Value Banana	Value Candy

Powershell Fundamentals

Powershell Objects

#Pipelines (|)

Each pipeline operator sends the results of the preceding command to the next command.
The output of the first command can be sent for processing as input to the second command. And that output can be sent to yet another command.

Example.

Notepad.exe
Get-Process notepad | Stop-Process

#One-Liner

One-liner is one continuous pipeline and not necessarily a command that's on one physical line. Not all commands that are on one physical line are one-liners.

Conditional Statments
(Using if, else, and elseif)

In PowerShell, conditional statements (`if`, `elseif`, and `else`) are used to control the flow of a script by executing different code blocks depending on the evaluation of conditions.

If statements

Syntax for a basic if statement

```
if (condition) {
        # Code to execute if condition is true
} elseif (another_condition) {
        # Code to execute if the first condition is false but this one is true
} else {
        # Code to execute if none of the conditions are true
}
```

Example of a basic if statement

```
If ($a -eq 1) {
Write-output "A is equal to 1"
} else ($a -ne 1) {$a++
} elseif ($a -eq 1) { write-output "a is not equal to 1"
}
```

Powershell Fundamentals

Loops

For Loops and Foreach Loops

#When should you use a forloop vs a foreach loop?
A forloop is best used when you are counting from an array or variable e.g looping 10 times from the variable of 1.

A foreach loop is best used when working with a list of information in an array or hash table.

#How to use a Forloop

Syntax
For ($<integer value> = <starting value> ; $<condition for the loop to end> ; <Repeater> {
<statement list>
}

Example
$numbers = 1
For ($i =0; $i = $numbers -lt 10; $i++)

What dose this mean?
Our variable ($numbers
}= 1) the int will check are $numbers to see if its less then (lt) then 10, then increment by 1.

```
For ($i =0; $i = $numbers -lt 10; $i++) {
Write-Output “Loop Number $i”
}
```

Powershell Fundamentals

Regular Expressions (REGEX)

Common Metacharacters and their meaning

Metacharacter	Meaning	Example
`^`	Start of a string/line anchor	`^Hello` → Matches `"Hello"` only at the beginning of the string.
`[`	Starts a character class	`[a-z]` → Matches any lowercase letter from `a` to `z`.
`.`	Matches any character (except newline by default)	`a.b` → Matches `"acb"`, `"a1b"`, `"a-b"`, etc.
`$`	End of a string/line anchor	`end$` → Matches `"end"` only at the end of the string.
`{`	Starts a quantifier (min/max occurrences)	`a{2,4}` → Matches `"aa"`, `"aaa"`, or `"aaaa"`.
`*`	Zero or more occurrences of the previous character/group	`go*d` → Matches `"gd"`, `"god"`, `"good"`, `"goood"`, etc.
`(`	Starts a capturing group	`(abc)+` → Matches `"abc"`, `"abcabc"`, `"abcabcabc"`, etc.
`\`	Escape character (used before special characters)	`\.` → Matches a literal dot (`.`) instead of "any character".
`+`	One or more occurrences of the previous character/group	`ba+` → Matches `"ba"`, `"baa"`, `"baaa"`, but not `"b"`.
`)`	Ends a capturing group	`(\d{3})` → Captures exactly three digits as a group.
`|`	`|`	OR operator (alternation)
`?`	Zero or one occurrence (makes previous character optional)	`colou?r` → Matches `"color"` and `"colour"`.
`< >`	Not typically used in standard regex (but can be used in some regex flavors for things like word boundaries in XML or HTML).	`<tag>` → In some regex flavors, can match XML/HTML tags.

Powershell Fundamentals

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

Using Regex and strings

#How to search a document and get a count of a a string of values
Get-Content .\<File Name> | Select-String <-Methods> “Regex expression” | % {$_.Match.Value} | Measure-Object

Methods
-Pattern
Used with `Select-String`, `-Match`, `-NotMatch`
Purpose: `Specifies a regular expression (regex) pattern to match in a string.`

-include
Used With: `Get-ChildItem` (`gci` or `ls`)
Purpose: Filters results to include only files or items that match the given pattern(s).

-Exclude
Used With: `Get-ChildItem`, `Copy-Item`, `Remove-Item`
Purpose: `Excludes files or items that match the given pattern(s).`

-NotMatch
Used With: Filtering conditions (`Where-Object`)

Powershell Fundamentals

Purpose: Returns objects where a pattern is **not** found in a string.

-Match
Used With: String comparisons
Purpose: Returns **True** if a string matches the specified pattern (regex).

-All Matches
Used With: `-Match`, `Select-String`
Purpose: Finds **all** matches in a string (instead of just the first one).

Powershell Fundamentals

If, elseif, and else statement example

Make a variable x and set it equal to 1. Make a statement to check x. If x is greater than one, print “x is greater than 1”. If x is less than one, print “x is less than 1”. If x is equal to 1, print “x is equal to one”

```
$x = 1
If ($x -eq 1){
Echo “x is equal to 1”
}
Elseif ($x -lt 1){
Echo “x is less than 1”
}
Elseif ($x -gt 1){
Echo “x is greater than 1”
}
Else {
Echo “how did you get here”
}
```

#If, elseif, else statement example built as a function

```
function grade {if ($studnetgrade -lt 69){
>> echo "Retrain"
>> }
>> elseif ($studnetgrade -gt 70 -and $studnetgrade -lt 79){
>> echo "you passed but barely."
>> }
>> elseif ($studnetgrade -gt 80 -and $studnetgrade -lt 89){
>> echo "Nice job you passed"
>> }
>> elseif ($studnetgrade -gt 90 -and $studnetgrade -lt 101){
>> echo "Hell ya cruched it"
>> }
>> else {
>> echo "need more if, elseif, else practise"
>> }
>> }
```

Powershell Fundamentals

Array Example

Make an array call $myarray with the value “dog”, “cat”, and “man”. Make a for each loop that will loop through the $myarray, at each loop, print “the value at this point is $value”.
– Turn it into a function called “get_the_function” and call the function.

1. Make your array

```
$myarray = @(“cat”, “dog”, “man”)
```

2. Make your foreach loop

```
Foreach ($value in $myarray){
Echo “the value at this point is $value”
}
```

3. Make it a function

```
Function “get_the_function” {
Foreach ($value in $myarray){
Echo “the value at this point is $value”
}
}
```