

# Manipulating Data within PowerShell Functions

---



**Liam Cleary**

CEO / MICROSOFT MVP / MICROSOFT CERTIFIED TRAINER

@shareplicity [www.shareplicity.com](http://www.shareplicity.com) | @helloitsliam [www.helloitsliam.com](http://www.helloitsliam.com)



# Overview



**Converting and Formatting Data Values**

**Manipulating String Data**

**Working with Custom Object Data**

**Loading and Iterating XML and JSON**



# Converting and Formatting Data Values

---

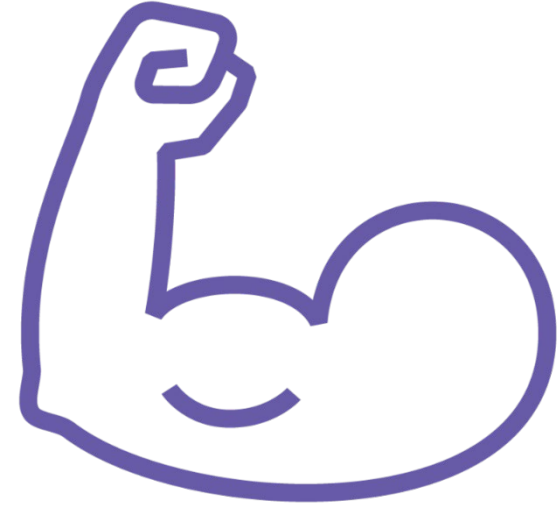


# Types of Variables



**Loosely Typed**

**When a value is assigned to an  
undefined type of variable**



**Strongly Typed**

**Type is assigned to variable**



# Using Variables to Store Data



Can store all types of values in PowerShell variables



A variable is a unit of memory in which values are stored



You declare variables by using a dollar sign (\$) before the name



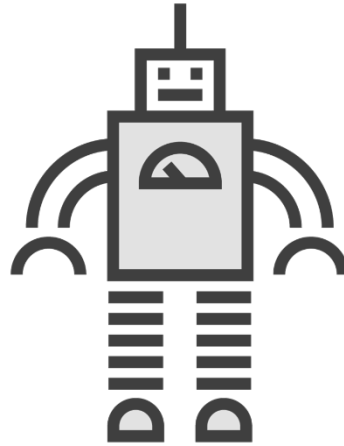
Variable names aren't case-sensitive, and can include spaces and special characters



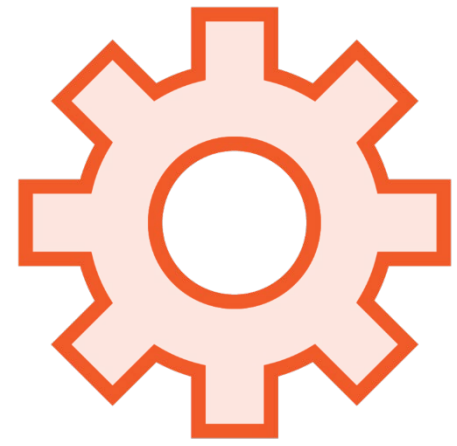
# Variable Types



**User-Created  
Variables**



**Automatic  
Variables**



**Preference  
Variables**

# User-created Variables

User-created variables are created and maintained by the user. The variables created within the PowerShell command-line exist only while the PowerShell window is open.



# Automatic Variables

Automatic variables store the state of PowerShell. PowerShell creates these variables and changes their values as required to maintain their accuracy.





# Preference Variables

Preference variables store user preferences for PowerShell. These variables are created by PowerShell and populated with default values.



# Creating User Variables

## # Create a basic variable

```
$variable1 = 1, 2, 3  
$variable2 = "C:\Documents\  
$variable3 = "January 1, 2021"
```

## # Create a typed variable

```
[Int]$variable1 = 10  
[DateTime]$variable2 = "January 1, 2021"
```



# Common Variable Data Types



Boolean



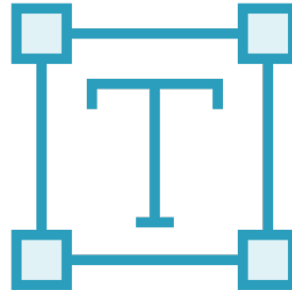
Date Time



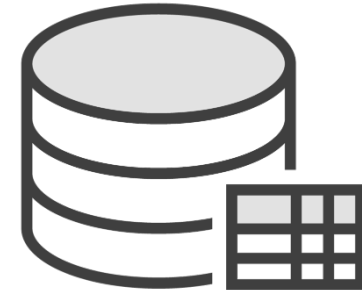
PowerShell Object



Script Block

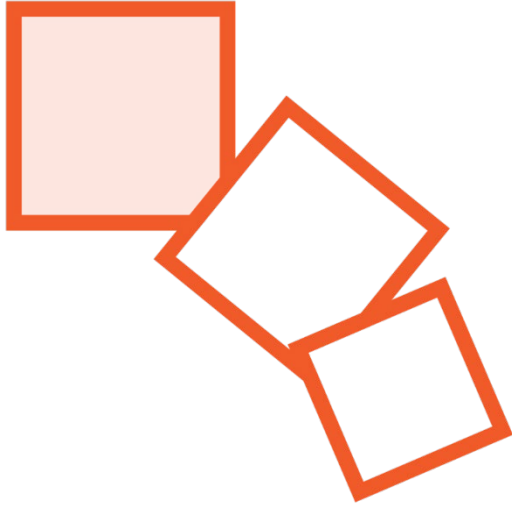


String



Array

# Casting Data Values



Converting one object type to another



Not all objects can be cast



# Casting Variables and Values

**# Variables**

```
$variable1 = "1"
```

```
$variable2 = "01/01/2021"
```

**# Converting String Variable to Integer**

```
[Int]$variable1
```

**# Converting String Variable to Date**

```
[DateTime]$variable2
```

**# Converting False Value to Integer**

```
[int]$false
```



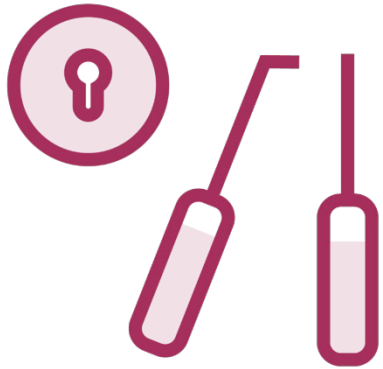
When a **VALUE** is cast to a particular datatype, it is a one-time change



When a **VARIABLE** is cast to a particular datatype it stays unless it is updated



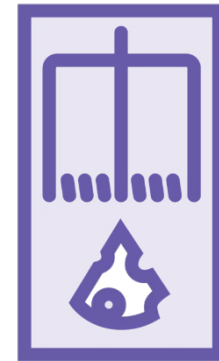
# Casting Using the -AS Operator



Use to test a  
conversion



Define the type after  
the variable



Can have unexpected  
results





# Casting Using the -AS Operator

## # Variables

```
$variable1 = "1"
```

```
$variable2 = "01/01/2021"
```

## # Cast String Variable to Integer

```
$variable1 -as [Int]
```

## # Cast String Variable to Date

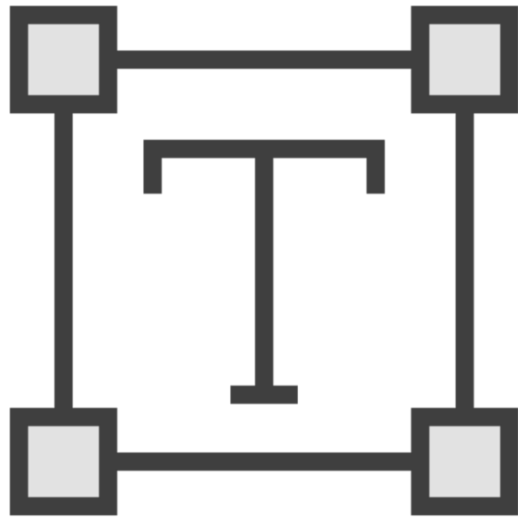
```
$variable2 -as [DateTime]
```

## # Cast False Value to Integer

```
$false -as [Int]
```



# Using the -F Operator



**Used to format a string expression**

**Supports complex formatting**

**Begin statements with selected format**

# Formatting Data Values using -F Operator

## # Variables

```
$variable1 = 123.4567890
```

```
$variable2 = 4503457892
```

## # Display Only three Decimal Places

```
"{0:n3}" -f $variable1
```

## # Add Spaces for Phone Number

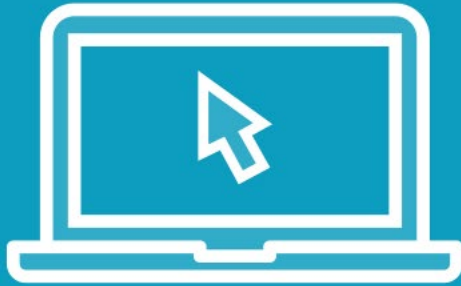
```
"{0:###-###-####}" -f $variable2
```

## # Display Year Only from Date

```
"{0:yyyy}" -f (Get-Date)
```



# Demo



## How to Convert and Format Values

- Formatting Values
- Cast Values to a different Type
- Cast using the -AS Operator

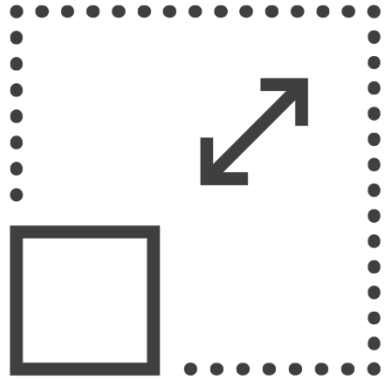


# Manipulating String Data

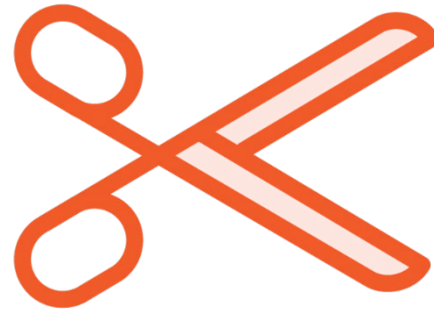
---



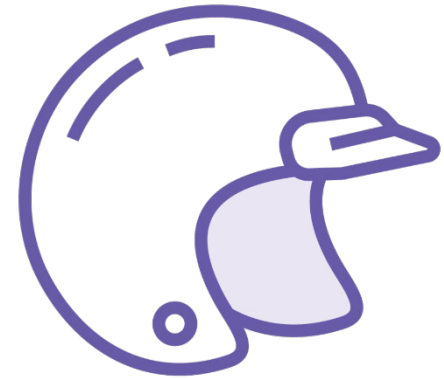
# Common String Manipulation



Replacing String  
Values



Splitting String  
Values

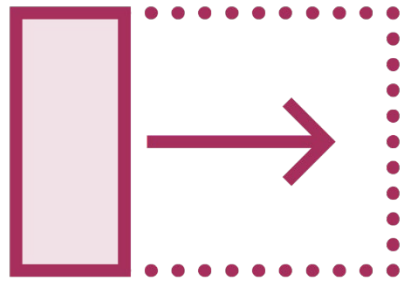


Padding String  
Values

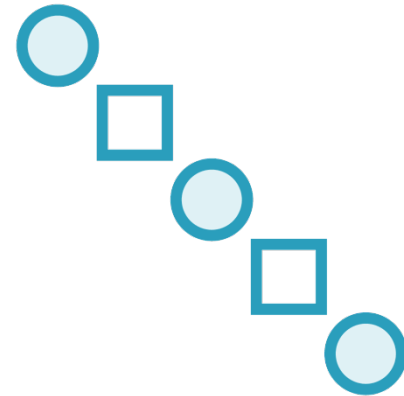
# Using the -Replace Operator



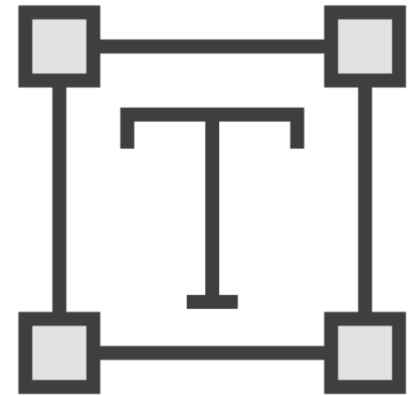
Input String



Replace  
Operator



Matching  
Pattern



Replacement  
String

# Using the -Replace Operator

## # Variable for Sentence

```
$variable1 = "The class instructor asked for a volunteer for a demonstration"  
$variable2 = "Jones Tom"
```

## # Read Variable and Replace Value

```
$variable1 -replace "instructor","teacher"
```

## # Read Variable, Replace Value and Load into New Variable

```
$replacevariable = $variable1 -replace "instructor","teacher"
```

## # Using Replace and RegEx to Swap Names

```
$variable2 = $variable2 -replace "([a-z]+\s[a-z]+)", '$2, $1'
```

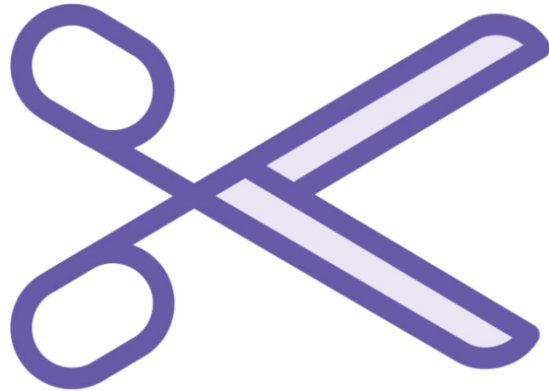
## # Using Replace and RegEx to Remove Spaces

```
$variable1 -replace '[^a-z]'
```





# Using the -Split Operator



**Splits one or more strings into substrings**

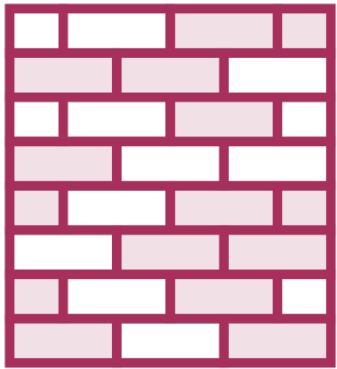
**Default split delimiter is whitespace**

**Other delimiters are characters, strings, patterns, or script blocks**

**All substrings return by default**



# The -Split Operator Syntax



Delimiter



Number of Substrings



Conditions the  
Delimiter Matches



# The -Split Operator Syntax

-Split <String>

-Split (<String[]>)

<String> -Split <Delimiter>[,<Max-substrings>[, "<Options>"]]

<String> -Split {<ScriptBlock>} [,<Max-substrings>]



# Using the -Split Operator

**# Split String Value using Default Delimiter**

```
-split "January February March April May June"
```

**# Split String Value using Comma Delimiter**

```
"January,February,March,April,May,June" -split ","
```

**# Split String Value into three using Comma Delimiter**

```
"January,February,March,April,May,June" -split ",",3
```

**# Split Variable Value using Comma Delimiter**

```
$variable = "January,February,March,April,May,June"
```

```
$variable -split ","
```



# Using the .Split() Function



**Splits input into multiple substrings based on the delimiters**

**Uses whitespace characters like space, tabs, and line-breaks as default delimiter**

**Available for all [String] type variables and values**



# Using the .Split() Function

**# Split Variable Value using Comma Delimiter**

```
$variable = "January,February,March,April,May,June"  
$variable.Split(',')
```

**# Nested Split Variable Value using Comma Delimiter**

```
$variable = "January,February,March,April;May;June"  
$variable.Split(',').Split(';')
```



# Padding String Values



Pad Left



Pad Right

# Padding String Values



## Pad Left

- Add padding to the left of the value
- Set the specified length

## Pad Right

- Add padding to the right of the value
- Set the specified length



# Padding String Values

Variable / Value	Width: Value Length + Padding	Padding Character
"Demonstration"	$13 + 1 = 14$	'A'
"Sample"	$6 + 1 = 7$	'B'
1	$1 + 5 = 6$	'O'
<code>\$variable</code>	$\text{$variable.Length} + 5 = X$	'T'



# Padding String Values

## # Standard Syntax

```
.PadLeft([Int]Width [,Padding Character])  
.PadRight([Int]Width [,Padding Character])
```

## # Padding the Left of a Value

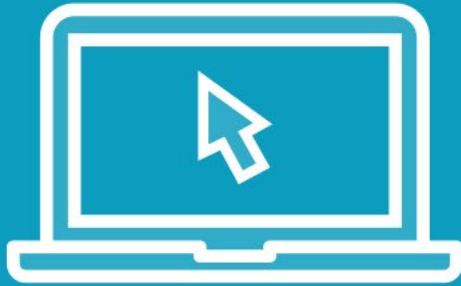
```
$variable = "Demonstration"  
$variable.PadLeft(14, 'A')
```

## # Padding the Right of a Value

```
$variable = "Demonstration"  
$variable.PadRight(14, 'B')
```



# Demo



## Manipulating String Data

- Replacing String Values
- Splitting String Values
- Padding String Values

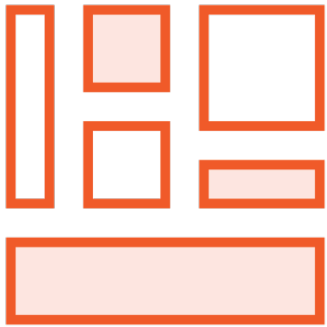


# Working with Custom Object Data

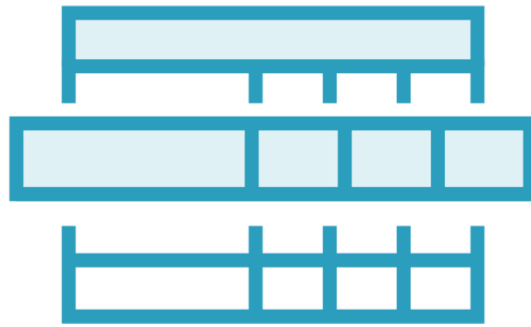
---



# Custom Object Data



Arrays



Hash Tables



Custom Object

# What are Arrays?

An Array is a list or collection of values or objects.  
Arrays only contain values not properties.



# Creating Arrays

Empty Arrays can  
be created by using  
"@()"

Comma separated  
lists can also create  
Arrays

The "Write-Output"  
command can  
create string Arrays



# Creating an Array

**# Create Empty Array**

```
$variable = @()
```

**# Create Array with Text Values**

```
$variable = @('January', 'February', 'March', 'April', 'May', 'June')
```

**# Create Array with Text Values not using "@()"**

```
$variable = 'January', 'February', 'March', 'April', 'May', 'June'
```

**# Create Array with Text Values not using "@()"**

```
$variable = Write-Output January February March April May June
```





# Retrieving Array Items

**# Create Array with Text Values**

```
$variable = @('January', 'February', 'March', 'April', 'May', 'June')
```

**# Access Array Item using Offset**

```
$variable[0]
```

**# Access Array Items using Multiple Offsets**

```
$variable[0,1,4]
```

**# Access Array Items using Range Operator as Offset**

```
$variable[2..5]
```



# Iterating Array Items

## # Create Array with Text Values

```
$variable = @('January', 'February', 'March', 'April', 'May', 'June')
```

## # Access Array Items using Pipeline with ForEach-Object Loop

```
$variable | ForEach-Object {"The month is: $PSItem"}
```

## # Access Array Items using a ForEach Loop

```
foreach ($item in $variable) {"The month is: $item"}
```

## # Access Array Items using the ForEach Method

```
$variable.ForEach({"The month is: $PSItem"})
```

## # Access Array Items using For Loop

```
for ($item = 0; $item -lt $variable.count; $item++) {  
    "The month is: {0}" -f $variable[$item];  
    Write-Host "Current Position: $item"  
}
```



# What are Hashtables?

A Hashtable is a data structure like an Array, except you store every value (object) using a key. Hashtables only contain values not properties. They are basic key/value stores.



# What are Hashtables?



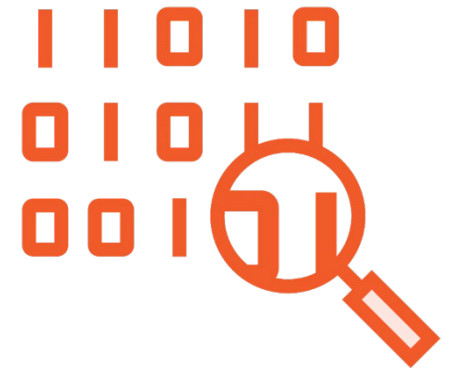
Also known as a dictionary or associative array

K	V

Data structure that stores one or more key/value pairs

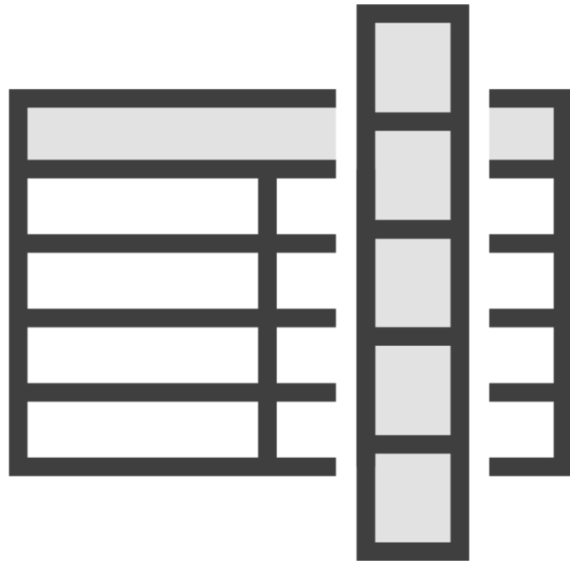


Keys and value in hash tables are .NET objects



Efficient for finding and retrieving data

# Creating Hashtables



Start the hash table with an at-sign (@)

Wrap the hash table in braces ({} )

Use an equal sign (=) to separate each key from its value

Use a semicolon (;) or a line break to separate the key/value pairs

# Creating a Hashtable

**# Create an Empty Hashtable**

```
$variable = @{ }
```

**# Create a Hashtable with Keys and Values**

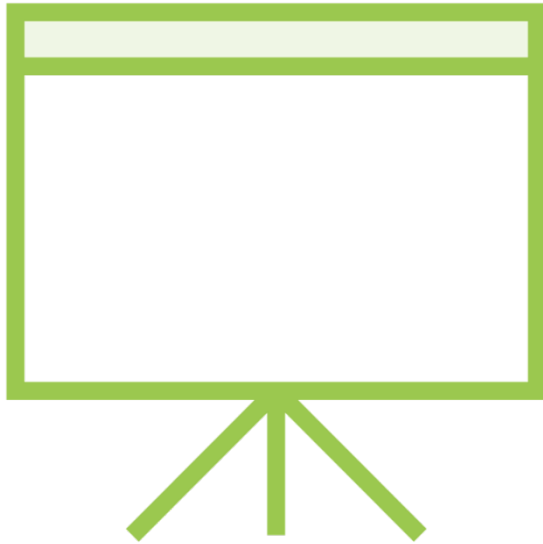
```
$variable = @{ Month = 5; Name = "May"; Season = "Spring" }
```

**# Create an Ordered Hashtable with Keys and Values**

```
$variable = [ordered]@{ Month = 5; Name = "May"; Season = "Spring" }
```



# Display Values from Hashtables



## Store Hashtable in variable

- Type `$variable` name, press Enter

## Use the "." notation to display all the keys or all the values

- Type `$variable.keys`, press Enter
- Type `$variable.values`, press Enter

## Each key name is also a property

- Type `$variable.Month`, press Enter
- Type `$variable["Month"]`, press Enter

# Iterating a Hashtable

## # Create a Country Population Hashtable

```
$variable = @{  
    Chine = 1439323776;  
    India = 1380004385;  
    America = 331002651;  
    Spain = 46754778  
}
```

## # Iterate all Keys and Values using ForEach-Object Loop

```
$variable.keys | ForEach-Object{  
    $output = '{0} has a population of {1}' -f $_, $variable[$_];  
    Write-Output $output  
}
```

## # Iterate all Keys and Values using For Loop

```
foreach($key in $variable.keys) {  
    $output = '{0} has a population of {1}' -f $key, $variable[$key];  
    Write-Output $output  
}
```





# What are PSCustomObject's?

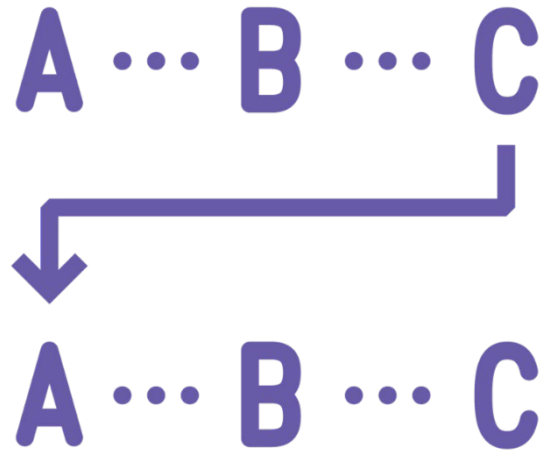
PSCustomObject's are a simple way to create structured data. A PSCustomObject is comprised of properties and values.



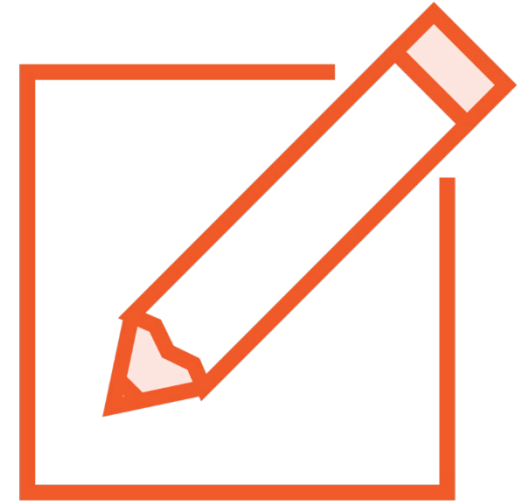
# PSCustomObject's



Like Hashtables,  
except strongly typed  
as "PSCustomObject"



Properties are ordered  
by default



PSCustomObject  
contains  
"NoteProperty", the  
same common  
PowerShell Objects

# Creating a PSCustomObject

**# Create an Empty PSCustomObject**

```
$variable = [PSCustomObject]@{ }
```

**# Create an Empty PSCustomObject**

```
$variable = New-Object -TypeName PSObject
```

**# Create and Populate a PSCustomObject**

```
$variable = [PSCustomObject]@{  
    'China' = '1439323776';  
    'India' = '1380004385';  
    'America' = '331002651';  
    'Spain' = '46754778';  
}
```

**# Add Items to PSCustomObject**

```
$variable | Add-Member -MemberType NoteProperty -Name 'Russia' -Value '145934462'  
$variable | Add-Member -MemberType NoteProperty -Name 'Norway' -Value '5421241'
```



# Retrieving PSCustomObject Properties and Values

**# Create and Populate a PSCustomObject**

```
$variable = [PSCustomObject]@{  
    'China' = '1439323776';  
    'India' = '1380004385';  
    'America' = '331002651';  
    'Spain' = '46754778';  
}
```

**# Return all PSCustomObject Key/Value Pairs**

```
$variable
```

**# Return Spain PSCustomObject**

```
$variable.Spain
```



# Testing PSCustomObject's



**-IS Operator**



**-ISNOT Operator**



# Testing PSCustomObject's

```
$variable1 = 10
```

```
$variable2 = "10"
```

```
$variable1 -is [Int]
```

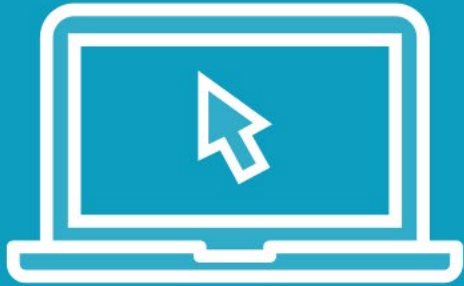
```
$variable1 -is $variable2.GetType()
```

```
$variable1 -isnot [Int]
```

```
$variable1 -isnot $variable2.GetType()
```



# Demo



## Create Custom Data Objects

## Retrieve Data from Custom Objects

- Load and Reuse Custom Object
- Test Objects using -IS Operator



# Loading and Iterating XML and JSON

---





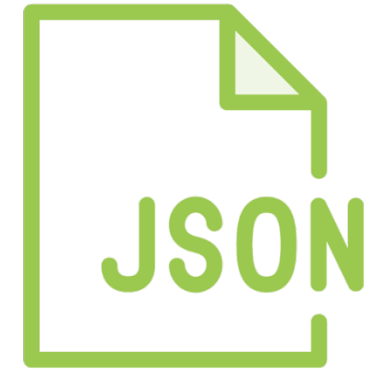
# Common Commands



ConvertTo-Xml



ConvertFrom-Json



ConvertTo-Json

# Loading XML



**Simplest approach to use SELECT-XML command**

**Uses XPath queries to search for text in XML strings and documents**

**Requires the PATH of the XML File and the XPATH within the XML document to search**

# Loading XML Data

## # Define Variables

```
$path = "C:\Documents\Data\Countries.xml"  
$xpath = "/Countries/Country/Name"
```

## # Load Xml

```
$xml = Select-Xml -Path $path -XPath $xpath
```



# Iterating and Retrieving XML Data

## # Define Variables

```
$path = "C:\Documents\Data\Countries-with-Attributes.xml"  
$xpath = "/Countries/Country"
```

## # Load Xml

```
$xml = Select-Xml -Path $path -XPath $xpath
```

## # Loop through Xml Property Values

```
$xml | ForEach-Object {$_ .Node.name}
```

## # Create Xml Variable and Load Data, then Retrieve Values

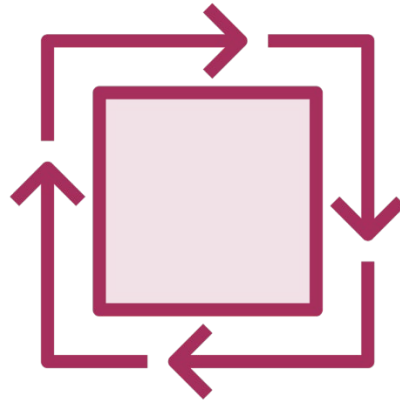
```
[xml]$xml = Get-Content -Path $path  
$xml.Countries.Country.Name  
$xml.Countries.Country.Population
```



# Steps for Iterating through XML Data



Read File and Cast to  
XML Object



Iterate through XML  
Data



Return Results



# Example Script

## # Define Variables

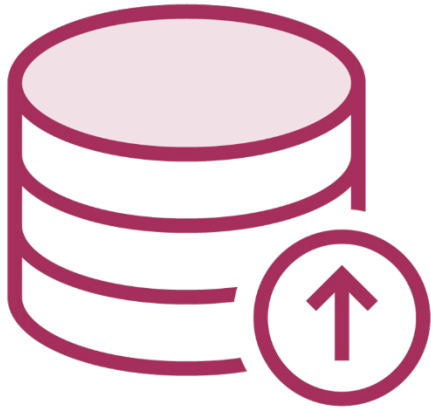
```
$path = "C:\Documents\Data\Countries-Checked.xml"
```

## # Load Xml

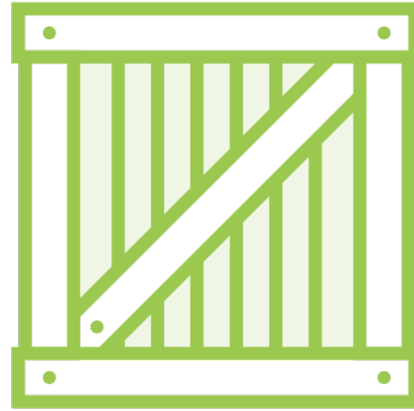
```
[xml]$xml = Get-Content -Path $path
$xml.Countries.Country | Where-Object Validated -eq 'True' | `
ForEach-Object {
    [PSCustomObject]@{
        Name = $_.Name
        Population = $_.Population
        Checked = $_.Validated
    }
}
```



# Working with JSON Data



Importing JSON



Exporting JSON



Testing JSON



# Import, Export and Test JSON

## # Variables

```
$path = "C:\Documents\Data\Countries.json"
```

## # Import JSON File

```
$json = Get-Content -Path $path | ConvertFrom-JSON
```

## # Export JSON File

```
$json | ConvertTo-JSON | Out-File $path
```

## # Test JSON File

```
Get-Content -Path $path -Raw | Test-JSON
```





# Iterating and Retrieving JSON Data

## # Variables

```
$path = "C:\Documents\Data\Countries.json"
```

## # Load JSON File

```
$json = (Get-Content -Path $path) | ConvertFrom-JSON  
$json.Countries
```

## # Loop JSON Data

```
Foreach ($item in $json)  
{  
    $item.Countries.Country | Select-Object Name, Population  
}
```

## # Loop JSON Data using Expand

```
Foreach ($item in $json)  
{  
    $item.Countries | Select-Object -ExpandProperty Country | `  
    Select-Object Name, Population  
}
```



# Retrieving JSON from Restful API

## # Variables

```
$uri = "https://swapi.dev/api/people/"
```

## # Load JSON File

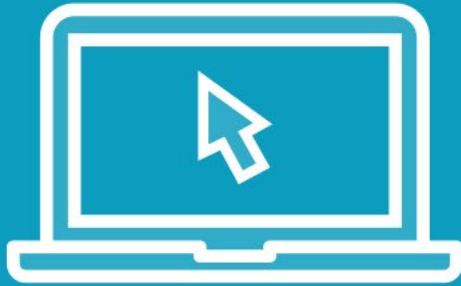
```
$json = Invoke-RestMethod -Uri $uri
```

## # Loop JSON Data

```
Foreach($item in $json.results)  
{  
    $item | Select-Object Name, Height, Gender  
}
```



# Demo



Load XML Data

Load JSON Data

Iterate XML and JSON Data



# Summary



Looked at how to Convert and Format Data Values

Manipulated String Data using Various Approaches

Worked with Custom Object Data

Loaded and then Iterated both XML and JSON Data and Files



# Up Next:

## Managing Errors and Exceptions in PowerShell

---

