PowerShell is the Windows scripting language and configuration management framework from Microsoft built on the .NET Framework. Windows 7 and up ship with PowerShell.
Nearly all examples below can be a part of a shell script or executed directly in the shell.

A key difference with Bash is that it is mostly objects that you manipulate rather than plain text.

Read more here.

If you are uncertain about your environment:

```
Get-ExecutionPolicy -List
Set-ExecutionPolicy AllSigned
# Execution policies include:
# - Restricted: Scripts won't run.
# - RemoteSigned: Downloaded scripts run only if signed by a trusted publisher.
# - AllSigned: Scripts need to be signed by a trusted publisher.
# - Unrestricted: Run all scripts.
help about_Execution_Policies # for more info

# Current PowerShell version:
$PSVersionTable
```

Getting help:

```
# Find commands
Get-Command about_* # alias: gcm
Get-Command -Verb Add
Get-Alias ps
Get-Alias -Definition Get-Process

Get-Help ps | less # alias: help
ps | Get-Member # alias: gm

Show-Command Get-EventLog # Display GUI to fill in the parameters

Update-Help # Run as admin
```

The tutorial starts here:

```
# As you already figured, comments start with #

# Simple hello world example:
echo Hello world!
# echo is an alias for Write-Output (=cmdlet)
# Most cmdlets and functions follow the Verb-Noun naming convention

# Each command starts on a new line, or after a semicolon:
echo 'This is the first line'; echo 'This is the second line'

# Declaring a variable looks like this:
$aString="Some string"
# Or like this:
$aNumber = 5 -as [double]
$aList = 1,2,3,4,5
```

```
$aString = $aList -join '--' # yes, -split exists also
$aHashtable = @{name1='val1'; name2='val2'}

# Using variables:
echo $aString
echo "Interpolation: $aString"
echo "`$aString has length of $($aString.Length)"
echo '$aString'
echo @"
This is a Here-String
$aString
"@
# Note that ' (single quote) won't expand the variables!
# Here-Strings also work with single quote

# Builtin variables:
# There are some useful builtin variables, like
echo "Booleans: $TRUE and $FALSE"
echo "Empty value: $NULL"
echo "Last program's return value: $?"
echo "Exit code of last run Windows-based program: $LastExitCode"
echo "The last token in the last line received by the session: $$"
echo "The first token: $^"
echo "Script's PID: $PID"
echo "Full path of current script directory: $PSScriptRoot"
echo 'Full path of current script: ' + $MyInvocation.MyCommand.Path
echo "FUll path of current directory: $Pwd"
echo "Bound arguments in a function, script or code block: $PSBoundParameters"
echo "Unbound arguments: $($Args -join ', ')."
# More builtins: `help about_Automatic_Variables`

# Inline another file (dot operator)
. .\otherScriptName.ps1


### Control Flow
# We have the usual if structure:
if ($Age -is [string]) {
    echo 'But.. $Age cannot be a string!'
} elseif ($Age -lt 12 -and $Age -gt 0) {
    echo 'Child (Less than 12. Greater than 0)'
} else {
    echo 'Adult'
}

# Switch statements are more powerfull compared to most languages
$val = "20"
switch($val) {
  { $_ -eq 42 }           { "The answer equals 42"; break }
  '20'                    { "Exactly 20"; break }
  { $_ -like 's*' }       { "Case insensitive"; break }
  { $_ -clike 's*'}       { "clike, ceq, cne for case sensitive"; break }
  { $_ -notmatch '^.*$'}  { "Regex matching. cnotmatch, cnotlike, ..."; break }
  { 'x' -contains 'x'}    { "FALSE! -contains is for lists!"; break }
  default                 { "Others" }
}

# The classic for
for($i = 1; $i -le 10; $i++) {
  "Loop number $i"
}
# Or shorter
1..10 | % { "Loop number $_" }

# PowerShell also offers
foreach ($var in 'val1','val2','val3') { echo $var }
# while () {}
# do {} while ()
# do {} until ()

# Exception handling
try {} catch {} finally {}
try {} catch [System.NullReferenceException] {
    echo $_.Exception | Format-List -Force
}


### Providers
# List files and directories in the current directory
ls # or `dir`
cd ~ # goto home

Get-Alias ls # -> Get-ChildItem
# Uh!? These cmdlets have generic names because unlike other scripting
# languages, PowerShell does not only operate in the current directory.
cd HKCU: # go to the HKEY_CURRENT_USER registry hive

# Get all providers in your session
Get-PSProvider


### Pipeline
# Cmdlets have parameters that control their execution:
Get-ChildItem -Filter *.txt -Name # Get just the name of all txt files
# Only need to type as much of a parameter name until it is no longer ambiguous
ls -fi *.txt -n # -f is not possible because -Force also exists
# Use `Get-Help Get-ChildItem -Full` for a complete overview

# Results of the previous cmdlet can be passed to the next as input.
# `$_` is the current object in the pipeline object.
ls | Where-Object { $_.Name -match 'c' } | Export-CSV export.txt
ls | ? { $_.Name -match 'c' } | ConvertTo-HTML | Out-File export.html

# If you get confused in the pipeline use `Get-Member` for an overview
# of the available methods and properties of the pipelined objects:
ls | Get-Member
Get-Date | gm
```

```
# ` is the line continuation character. Or end the line with a |
Get-Process | Sort-Object ID -Descending | Select-Object -First 10 Name,ID,VM `
    | Stop-Process -WhatIf

Get-EventLog Application -After (Get-Date).AddHours(-2) | Format-List

# Use % as a shorthand for ForEach-Object
(a,b,c) | ForEach-Object `
    -Begin { "Starting"; $counter = 0 } `
    -Process { "Processing $_"; $counter++ } `
    -End { "Finishing: $counter" }

# Get-Process as a table with three columns
# The third column is the value of the VM property in MB and 2 decimal places
# Computed columns can be written more verbose as:
# `@{name='lbl';expression={$_}`
ps | Format-Table ID,Name,@{n='VM(MB)';e={'{0:n2}' -f ($_.VM / 1MB)}} -autoSize


### Functions
# The [string] attribute is optional.
function foo([string]$name) {
    echo "Hey $name, have a function"
}

# Calling your function
foo "Say my name"

# Functions with named parameters, parameter attributes, parsable documention
<#
.SYNOPSIS
Setup a new website
.DESCRIPTION
Creates everything your new website needs for much win
.PARAMETER siteName
The name for the new website
.EXAMPLE
New-Website -Name FancySite -Po 5000
New-Website SiteWithDefaultPort
New-Website siteName 2000 # ERROR! Port argument could not be validated
('name1','name2') | New-Website -Verbose
#>
function New-Website() {
    [CmdletBinding()]
    param (
        [Parameter(ValueFromPipeline=$true, Mandatory=$true)]
        [Alias('name')]
        [string]$siteName,
        [ValidateSet(3000,5000,8000)]
        [int]$port = 3000
    )
    BEGIN { Write-Verbose 'Creating new website(s)' }
    PROCESS { echo "name: $siteName, port: $port" }
    END { Write-Verbose 'Website(s) created' }
}


### It's all .NET
# A PS string is in fact a .NET System.String
# All .NET methods and properties are thus available
'string'.ToUpper().Replace('G', 'ggg')
# Or more powershellish
'string'.ToUpper() -replace 'G', 'ggg'

# Unsure how that .NET method is called again?
'string' | gm

# Syntax for calling static .NET methods
[System.Reflection.Assembly]::LoadWithPartialName('Microsoft.VisualBasic')

# Note that .NET functions MUST be called with parentheses
# while PS functions CANNOT be called with parentheses.
# If you do call a cmdlet/PS function with parentheses,
# it is the same as passing a single parameter list
$writer = New-Object System.IO.StreamWriter($path, $true)
$writer.Write([Environment]::NewLine)
$writer.Dispose()

### IO
# Reading a value from input:
$Name = Read-Host "What's your name?"
echo "Hello, $Name!"
[int]$Age = Read-Host "What's your age?"

# Test-Path, Split-Path, Join-Path, Resolve-Path
# Get-Content filename # returns a string[]
# Set-Content, Add-Content, Clear-Content
Get-Command ConvertTo-*,ConvertFrom-*


### Useful stuff
# Refresh your PATH
$env:PATH = [System.Environment]::GetEnvironmentVariable("Path", "Machine") +
    ";" + [System.Environment]::GetEnvironmentVariable("Path", "User")

# Find Python in path
$env:PATH.Split(";") | Where-Object { $_ -like "*python*"}

# Change working directory without having to remember previous path
Push-Location c:\temp # change working directory to c:\temp
Pop-Location # change back to previous working directory
# Aliases are: pushd and popd

# Unblock a directory after download
Get-ChildItem -Recurse | Unblock-File

# Open Windows Explorer in working directory
ii .
```

```
# Any key to exit
$host.UI.RawUI.ReadKey()
return

# Create a shortcut
$WshShell = New-Object -comObject WScript.Shell
$Shortcut = $WshShell.CreateShortcut($link)
$Shortcut.TargetPath = $file
$Shortcut.WorkingDirectory = Split-Path $file
$Shortcut.Save()
```

Configuring your shell

```
# $Profile is the full path for your `Microsoft.PowerShell_profile.ps1`
# All code there will be executed when the PS session starts
if (-not (Test-Path $Profile)) {
    New-Item -Type file -Path $Profile -Force
    notepad $Profile
}
# More info: `help about_profiles`
# For a more usefull shell, be sure to check the project PSReadLine below
```

Interesting Projects

- Channel9 PowerShell tutorials
- PSGet NuGet for PowerShell
- PSReadLine A bash inspired readline implementation for PowerShell (So good that it now ships with Windows10 by default!)
- Posh-Git Fancy Git Prompt (Recommended!)
- PSake Build automation tool
- Pester BDD Testing Framework
- Jump-Location Powershell `cd` that reads your mind
- PowerShell Community Extensions (Dead)

Not covered

- WMI: Windows Management Intrumentation (Get-CimInstance)

- Multitasking: Start-Job -scriptBlock {…},
- Code Signing
- Remoting (Enter-PSSession/Exit-PSSession; Invoke-Command)