Зміст

1. PS як платформа розв’язування задач по адмініструванню ОС Windows – переваги, недоліки та особливості застосування.

PowerShell одночасно є і оболонкою командного рядка (користувач може працювати в інтерактивному режимі) і середовищем виконання сценаріїв, які пишуться на спеціальній мові PowerShell. Всі команди в PowerShell мають детальну вбудовану довідку, підтримується функція автоматичного завершення назв команд і їх параметрів при введенні з клавіатури, для багатьох команд є псевдоніми.

В PowerShell окрему увагу приділено питанням безпеки при роботі зі сценаріями (наприклад, запустити сценарій можна тільки з зазначенням повного шляху до нього, а за замовчуванням запуск сценаріїв PowerShell в системі взагалі заборонений).

Мова PowerShell нескладна для вивчення, писати на ній сценарії, які звертаються до зовнішніх об'єктів досить легко. В цілому, оболонка PowerShell набагато зручніша і потужніша своїх попередників (cmd.exe і WSH).

Головною особливістю середовища PowerShell, що відрізняє її від всіх інших оболонок командного рядка, є те, що одиницею обробки і передачі інформації тут є об'єкт, а не рядок тексту. У командному рядку PowerShell висновок результатів команди є не текст (в сенсі послідовності байтів), а об'єкт (дані разом з властивими їм методами). В силу цього працювати в PowerShell стає простіше, ніж в традиційних оболонках, так як не потрібно виконувати ніяких маніпуляцій по виділенню потрібної інформації з символьного потоку.

Оболонка Windows PowerShell була задумана розробниками Microsoft як більш потужне середовище для написання сценаріїв та роботи з командного рядка. Мета - створити середовище складання сценаріїв, яка найкращим чином підходила б для сучасних версій операційної системи Windows і була б більш функціональною, що розширюється і простий у використанні, ніж будь-який аналогічний продукт для будь-якої іншої операційної системи. В першу чергу це середовище повинна була підходити для вирішення завдань, що стоять перед системними адміністраторами, а також задовольняти вимогам розробників програмного забезпечення, надаючи їм засоби для швидкої реалізації інтерфейсів управління створюваними додатками.

Для досягнення цих цілей були вирішені наступні завдання:

 Забезпечення прямого доступу з командного рядка до об'єктів COM, BMI і .NET. У новій оболонці присутні команди, що дозволяють в інтерактивному режимі працювати з COM-об'єктами, а також з екземплярами класів, перероблених в інформаційних схемах WMI і .NET.

 Організація роботи з довільними джерелами даних в командному рядку за принципом файлової системи. Наприклад, навігація по системного реєстру або сховища цифрових сертифікатів виконується з командного рядка за допомогою аналога команди cd інтерпретатора cmd.exe.

 Розробка інтуїтивно зрозумілій уніфікованої структури вбудованих команд, заснованої на їх функціональне призначення.

У новій оболонці імена всіх внутрішніх команд (в PowerShell вони називаються Командлети) відповідають шаблону "дієслово-іменник", наприклад, Get-Process (отримати інформацію про процес), Stop-Service (зупинити службу), Clear-Host (очистити екран консолі) і т. д. Для однакових параметрів внутрішніх команд використовуються стандартні

імена, структура параметрів у всіх командах ідентична, всі команди обробляються одним синтаксичним аналізатором.

 Забезпечення можливості розширення вбудованого набору команд. Внутрішні команди PowerShell можуть доповнюватися командами, які створюються користувачем. При цьому вони повністю інтегруються в оболонку, інформація про них може бути отримана з стандартної довідкової системи PowerShell.

 Організація підтримки знайомих команд з інших оболонок. У PowerShell на рівні псевдонімів власних внутрішніх команд підтримувати найбільш часто використовувані стандартні команди з оболонки cmd.exe і UNIX-оболонок.

 Розробка повноцінної вбудованої довідкової системи для внутрішніх команд. Для більшості внутрішніх команд в довідковій системі дано докладний опис і приклади використання. У будь-якому випадку вбудована довідка по будь-якій внутрішній команді буде містити короткий опис всіх її параметрів.

 Реалізація автоматичного завершення при введенні з клавіатури імен команд, їх параметрів, а також імен файлів і папок. Дана можливість значно спрощує і прискорює введення команд з клавіатури.

2.Регулярні вирази

Регулярні вирази (Regular Expressions, RegExp) - це спеціалізований мова для пошуку і обробки тексту. Основою регулярних виразів є керуючі символи (метасимволи), а саме регулярний вираз по суті є шаблоном, що визначає правила пошуку.

Регулярні вирази в різній мірі підтримуються в різних мовах \ середовищах програмування і навіть в деяких текстових редакторах. PowerShell базується на .NET, відповідно в ньому доступні практично всі можливості .NET, що стосуються регулярних виразів.

Оператор match

Для прикладу ми будемо використовувати оператор match, який є одним з основних інструментів для роботи з регулярними виразами в PowerShell. Він порівнює текст зліва з регулярним виразом справа, і якщо текст підходить під вираз, повертає True, якщо не підходить - False:

Метасимволи

Для позитивного результату регулярному виразу потрібно лише наявність відповідних символів, незалежно від їх розташування в тексті. Це поведінка за умовчанням, але його можна змінити за допомогою спеціальних метасимволов які дозволяють точно вказати на позицію в рядку, в якій повинні бути шукані символи.

^ - збігається з позицією на початку рядка, в розширеному режимі також збігається з позицією після будь-якого символу нового рядка;

\ A - не підтримує розширений режим і завжди відповідає початку рядка;

$ - збігається з позицією в кінці рядка і перед символом нового рядка, завершальним текст, в розширеному режимі збігається з позицією перед будь-яким символом нового рядка;

\ Z - не підтримує розширений режим, збігається з позицією в кінці рядка і перед завершальним символом нового рядка;

\ z - не підтримує розширений режим, збігається з позицією в кінці рядка тільки в тому випадку, якщо перед ним немає символу нового рядка.

Приклади

Метасимволи кришка ^ і \ A збігаються з початком рядка, перед першим символом. Для прикладу виведемо список служб, що починаються з символу "b":

Get-Service | where {$ _. name -match "^ b"}

Метасимволи долар $, \ Z або \ z збігаються з закінченням рядка. Для прикладу виведемо список служб, що закінчуються на "s":

Get-Service | where {$ _. name -match "s $"}

Межі

Як вже було сказано, для регулярного виразу немає необхідності в точному збігу, досить наявності шуканих символів в тексті. Так наприклад слово "cat" буде знайдено не тільки в "My cat is fat", але і в "bobcat", "category", "staccato" та інших, які не мають ні найменшого відношення до котячих. Тому, якщо необхідно тільки ціле слово, можна позначити його межі (boundaries) за допомогою метасимвола \ b.

3.Сценарії у PowerShell. Передача параметрів у сценарії.

Сценарій це блок коду, що зберігається в окремому файлі з розширенням .ps1. Це основна перевага сценаріїв, так як можна зберегти код, позбавивши себе від необхідності набирати його кожного разу вручну. Сценарії можна писати в будь-якому текстовому редакторі, головне, зберегти написане в файлі з розширенням .ps1. Сценарій можна написати і в консолі, скопіювавши потім текст в редактор і зберігши. Тим часом, в PowerShell вже є інтегроване середовище сценаріїв, що спрощує написання.

Передача параметрів.

Іноді створюється вами сценарій або функція повинна прийняти будь-яке вхідне значення - ім'я комп'ютера, шлях до папки, назва сервісу і т.п. У PowerShell є кілька способів передати дані в сценарій з командного рядка, зробивши їх введення більш простим і ефективним.

Найпростіший варіант передачі даних - використовувати вбудовану змінну $ args, яка має тип одновимірний масив (hashtable). Для цього створимо скрипт з ім'ям service.ps1 ось такого змісту:

Get-Service -Name $ args [0] -ComputerName $ args [1]

Цей скрипт повинен вивести стан заданого сервісу \ сервісів для одного або декількох комп'ютерів. Ім'я сервісу та комп'ютера передаються в скрипт в якості аргументів.Тепер виконаємо його, вказавши в якості аргументів сервіс друку (spooler) і ім'я комп'ютера SRV1:

. \ Service.ps1 spooler SRV1

Отримавши цю команду, PowerShell підставить замість $ args [0] назва сервісу, замість $ args [1] ім'я комп'ютера і виведе отриманий результат.

При використанні $ args кожне з згаданих значень додається в масив, в результаті ми отримуємо можливість передавати в сценарій будь-яку необхідну кількість параметрів.

4. Використання фільтрів у PowerShell

Фільтр - це функція особливого виду, яка, перебуваючи всередині конвеєра, запускається для кожного вхідного елемента (звичайні функції запускаються один раз для всієї сукупності елементів, сформованої попередньою командою конвеєра).

Синтаксично фільтри відрізняються від функцій лише тим, що замість ключового слова Function вказується слово Filter:

Filter ім'я_фільтра (параметри) {блок_коду}

Однак алгоритми роботи звичайної функції і фільтра відрізняються, якщо вони знаходяться всередині конвеєра. У звичайній функції доступ до вхідних елементів конвеєра здійснюється через колекцію $ Input. У фільтрі ж визначена змінна $ _, що відповідає поточному елементу конвеєра, який проходить через даний фільтр.

Розглянемо приклад. Напишемо фільтр, який буде подвоювати числа, які проходять через нього по конвеєру. Такий фільтр буде складатися всього з одного виразу:

```
PS C: \> Filter Double {$ _ * 2}
```

Перевіримо роботу даного фільтра:

```
PS C: \> 1..4 | Double
2
4
6
8
PS C: \> 1..4 | Double | Double
4
8
12
16
```

ЗАУВАЖЕННЯ

Функціональність фільтрів дуже схожа на функціональність командлети ForEach-Object, яку можна вважати безіменним фільтром.

5.Рекурсія в PowerShell

Рекурсія — виклик функції чи процедури з неї самої (звичайно з іншими значеннями вхідних параметрів) безпосередньо чи через інші функції (наприклад, функція A викликає функцію B, а функція B — функцію A). Кількість вкладених викликів функції чи процедури називається глибиною рекурсії.

Міць рекурсивного визначення об'єкта в тім, що таке кінцеве визначення здатне описувати нескінченно велике число об'єктів. За допомогою ж рекурсивної програми можливо описати нескінченне обчислення, причому без явних повторень частин програми.

Задача. Використовуючи засоби PoweShell розробити рекурсивну функцію для обчислення факторіала (n!) натурального числа. За її допомогою обчислити величину C_n^m

```
function Get-Factorial ($n) {
   if ($n -eq 0) {
      return 1
   }
   $fact = 1
   1..$n | ForEach { $fact *= $_ }
   return $fact
}

$n = 10
$m = 2

foreach ($i in 0..$n) {
   echo ("{0}! = {1}" -f $i,(Get-Factorial $i))
}

$C = ((Get-Factorial($n))/((Get-Factorial($m))*(Get-Factorial($n-$m))))
echo $C
```

6. Функції у PowerShell.

Функція в PowerShell – це блок коду, який має назву і знаходиться в пам'яті до завершення поточного сеансу командної оболонки. Якщо функція визначається без формальних параметрів, то для її задання достатньо вказати ключове слово Function, потім ім'я функції і список виразів, з яких складається тіло функції(даний список повинен бути у фігурних дужках). Наприклад створимо функцію MyFunc:

```
PS C:\> Function MyFunc{"Hello world!"}
PS C:\>
```

Для виклику цієї функції потрібно просто ввести її ім'я:

```
PS C:\> MyFunc
Hello world!
```

Відзначимо, що в багатьох мовах програмування при виконанні функції після її імені потрібно вказувати круглі дужки. У PowerShell цього робити не можна:

```
PS C:\> MyFunc()
Вираз очікується після '('.
В рядку:1 знак:8
+ MyFunc() <<<<
```

Функції можуть працювати з аргументами, які передаються їм при запуску,
причому підтримуються два варіанти обробки таких аргументів: за допомогою змінної $ Args і шляхом завдання формальних параметрів.

4.1 Обробка аргументів функцій за допомогою змінної $ Args

Функція в PowerShell має доступ до аргументів, з якими вона була запущена, навіть якщо при визначенні цієї функції не були задані формальні параметри. Всі аргументи, з якими була запущена функція, автоматично зберігаються в змінній $ Args. Іншими словами, в змінній $ Args міститься масив, елементами якого є параметри функції, зазначені при її запуску. Для прикладу додамо змінну $ Args в нашу функцію MyFunc:

```
PS C:\> Function MyFunc{"Hello, $Args!"}
```

Так як змінна $ Args поміщена в рядок в подвійних лапках, то при запуску функції значення цієї змінної буде обчислено (розширено), і результат буде вставлений в рядок. Викличемо функцію MyFunc з трьома параметрами:

```
PS C:\> MyFunc Nazar Nazar Nazar
Hello, Nazar Nazar Nazar!
```

Як бачите, три зазначених нами параметра (три елементи масиву $ Args) поміщені у вихідний рядок і розділені між собою пробілами. Можна, змінити символ, що розділяє елементи масивів при їх підстановці в рядки, перевизначивши значення спеціальної змінної $ OFS:

```
PS C:\> Function MyFunc{
>> $OFS=","
>> "Hello, $Args!"}
>>
PS C:\> MyFunc Nazar Nazar Nazar
Hello, Nazar,Nazar,Nazar!
```

Зверніть увагу, що на відміну від традиційних мов програмування, функції в PowerShell є командами (це не методи об'єктів!), Тому їх аргументи вказуються через пробіл без додаткових круглих дужок і виділення символьних рядків лапками. Щоб наочно переконатися в цьому, запустимо функцію MyFunc наступним чином:

```
PS C:\> MyFunc("Nazar","Nazar","Nazar")
Hello, System.Object[]!
```

Нагадаємо, що масиви в PowerShell задаються перерахуванням своїх елементів, а круглі дужки, що оточують який-небудь вираз, означають, що цей вислів має бути обчислено. Тому помилки при такому виклику функції не виникає, однак результат її виконання виявляється зовсім іншим, адже в даному випадку в функцію передаються не три символьних аргументи, а один аргумент, який є масивом з трьох елементів!

Так як змінна $ Args є масивом, то всередині можна вибрати окремим аргументів по їх порядковому номеру (нагадаємо, що нумерація елементів масивів починається з нуля), а за допомогою методу Count визначати загальну кількість аргументів, переданих функції. Для прикладу створимо функцію SumArgs, яка буде повідомляти про кількість своїх аргументів і обчислювати їх суму:

```
PS C:\> Function SumArgs{"Кількість аргументів: $($Args.Count)"
>> $n=0
>> For($i=0; $i -lt $Args.Count; $i++) { $n+=$Args[$i] }
>> "Сума аргументів: $n"}
>>
PS C:\>
```

Запустимо функцію SumArgs з трьома числовими аргументами:

```
PS C:\> SumArgs 1 2 3
Кількість аргументів: 3
Сума аргументів: 6
```

Крім використання масиву $ Args в PowerShell підтримується альтернативний підхід до обробки аргументів функцій - за допомогою завдання формальних параметрів.

7. Управляючі оператори та оператори циклу у PowerShell

У мові PowerShell, як і в будь-якій іншій алгоритмічній мові, є елементи, що дозволяють виконати логічне порівняння і зробити різні дії в залежності від його результату : або дають можливість повторювати одну або кілька команд знову і знову.

Інструкція if…elseif…else

Логічні порівняння лежать в основі практично всіх алгоритмічних мов програмування. У PowerShell за допомогою інструкції If можна виконувати певні блоки коду тільки в тому випадку, коли задана умова має значення True. Також можна задати одне або кілька додаткових умовних блоків. Відповідні їм умови будуть перевірятися, якщо всі попередні умови мали значення False. Нарешті, можна задати додатковий блок коду, який буде виконуватися в тому випадку, якщо жодна з умов не має значення True. Синтаксис інструкції if в загальному випадку має такий вигляд

if (умова1) {блок_коду1}
[elseif (умова2) {блок_коду2}]
[else {блок_коду3}]

Умовні вирази в powershell формуються, найчастіше, за допомогою операторів порівняння і логічних операторів. В якості умовних виразів можна використовувати конвейєри команд powershell. В оболонці powershell в інтерактивному режимі можна виконувати інструкції, що складаються з декількох рядків (це може бути дуже зручно при налагодженні сценаріїв).

Цикл While

У powershell підтримуються кілька видів циклів. Найпростіший з них цикл while, в якому команди виконуються до тих пір, поки умова, яка перевіряється, має значення True. Інструкція while має наступний синтаксис

while (умова) {блок команд}

При виконанні інструкції while оболонка powershell обчислює розділ *умова* інструкції, перш ніж перейти до розділу *блок команд*. Умова в інструкції приймає значення True або false. До тих пір, поки умова має значення true, powershell повторює виконання розділу *блок команд*. Як і в інструкції If, в умовному виразі циклу while може використовуватися конвеєр команд powershell. Розділ *блок команд* інструкції while містить одну або кілька команд, які виконуються кожен раз при вході в цикл і його повторенні.

Цикл Do…While

Цикл do while схожий на цикл while, однак умова в ньому перевіряється не до блоку команд, а після

do {блок команд} while (умова)

Цикл For

Інструкція for в powershell реалізує ще один тип циклів - цикл з лічильником. Зазвичай цикл for застосовується для проходження по масиву і виконання певних дій з кожним із його елементів. У powershell інструкція for використовується не так часто, як в інших мовах програмування, так як колекції об'єктів зазвичай зручніше обробляти за допомогою інструкції forEach. Однак якщо необхідно знати, з яким саме елементом колекції або масиву ми працюємо на цій ітерації, то цикл for може допомогти. Синтаксис інструкції for

for (ініціалізація; умова; повторення) {блок команд}

Складові частини циклу for мають наступний сенс: **ініціалізація** - це одна або кілька поділюваних комами команд, які виконуються перед початком циклу; **умова** - частина інструкції for, яка може приймати логічне значення True або false; **повторення** - це одна або кілька поділюваних комами команд, які виконуються при кожному повторенні циклу; **блок команд** - це набір з однієї або декількох команд, що виконуються при входженні в цикл або при його повторенні.

Цикл ForEach

Інструкція forEach дозволяє послідовно перебирати елементи колекцій. Найпростішим і найбільш часто використовуваним типом колекції, по якій проводиться переміщення, є масив. Зазвичай в циклі forEach одна або кілька команд виконуються на кожному елементі масиву. Особливістю циклу forEach є те, що його синтаксис і робота залежать від того, де розташована інструкція forEach: поза конвеєра команд або всередині конвеєра.

Інструкція ForEach поза конвеєром команд

В цьому випадку синтаксис циклу forEach має наступний вигляд

foreach ($елемент in $колеція) {блок команд}

У круглих дужках вказується колекція, по якій проводиться ітерація. При виконанні циклу forEach система автоматично створює змінну *елемент*. Перед кожною ітерацією в циклі цій змінній присвоюється значення чергового елемента в колекції. У розділі *блок команд* містяться команди, що виконуються на кожному елементі колекції.

Інструкція ForEach всередині конвеєра команд

Якщо інструкція foreach з'являється всередині конвеєра команд, то powershell використовує псевдонім foreach, відповідний командлету ForEach-Object. Тобто в цьому випадку фактично виконується командлет ForEach-Object і вже не потрібно вказувати частину інструкції, так як елементи колекції блоку команд надає попередня команда в конвеєрі. Синтаксис інструкції foreach, застосовуваної всередині конвеєра команд, в найпростішому випадку виглядає наступним чином

команда | ForEach-Object {блок команд}

Мітки циклів, іструкції Break і Continue

Інструкція Break дозволяє вийти з циклу будь-якого типу, не чекаючи закінчення його ітерацій.

```
$i = 0; while ($True) { if ($i++ -ge 3) { break } $i }
```

Інструкція continue здійснює перехід до наступної ітерації циклу будь-якого типу.

```
for ($i = 0; $i -le 5; $i++) { if ($i -ge 4) { continue } $i }
```

У мові powershell підтримується можливість негайного виходу або переходу до наступної ітерації не тільки для одиночного циклу, але і для вкладених циклів. Для цього циклам присвоюються спеціальні мітки, які вказуються на початку рядка перед ключовим словом, що задає цикл того чи іншого типу. Такі мітки потім можна використовувати у вкладених циклах спільно з інструкціями Break або continue, вказуючи, на який саме цикл повинні діяти в даній інструкції.

```
:outer while ($True) {
        while ($True) {
                break outer
        }
}
```

Інструкція Switch

Об'єднує декілька перевірок умов всередині однієї конструкції. У PowerShell володіє додатковими можливостями:

- може використовуватися як аналог циклу, перевіряючи значення не єдиного елементу, а цілого масиву;
- може перевіряти елементи на відповідність шаблону з підстановочними символами або регулярними виразами;
- може обробляти текстові файли, використовуючи в якості елементів для перевірки рядки з файлів.

Найпростіша форма інструкції switch:

```
$a = 3
switch ($a) {
        1 {'One'}
        2 {'Two'}
        3 {'Three'}
        4 {'Four'}}
```

Якщо для значення що перевіряється справедливі декілька умов то вони всі будуть виконані:

```
$a = 1
switch ($a) {
        1 {'One'}
        2 {'Two'}
        1 {'One again'}}
```

Якщо потрібно обмежетися тільки першим співпадінням, то слід використати інструкцію break:

```
$a = 1
switch ($a) {
        1 {'One'; break}
        2 {'Two'}
        1 {'One again'}}
```

Ключове слово default задає дію, що буде виконана у випадку коли не було жодного співпадіння:

```
switch (3) {
        1 {'One'}
        2 {'Two'}
        Default {'Not one and not two'}}
```

Для врахування регістру слід вказати параметр -CaseSensitive:

```
switch -CaseSensitive ('abc') {
        'abc' {'First match'}
        'ABC' {'Second match'}
}
```

Окрім звичайного порівнянн можна перевіряти елементи на відповідність шаблону з підстановочними символами, для цього використовується параметр -Wildcard:

```
switch -Wildcard ('abc') {
        'a*' {'Begins with a'}
        '*c' {'Ends with c'}
}
```

Об'єкт, що перевіряється, доступний всередині інструкції через спеціальну змінну $_:

```
switch -Wildcard ('abc') {'a*' {'$_ begins with a'}    }
```

8. Оператори порівняння та логічні оператори.

Оператори порівняння

Оператори порівняння дозволяють порівнювати значення параметрів в командах. При кожній операції порівняння створюється умова, в залежності від виконання або невиконання якого оператор приймає або значення $True (істина), або значення $False (брехня). Основні оператори порівняння наведені в таблиці.

Оператор	**Значення**	**Приклад (вертає значення true)**
-eq –ceq –ieq	Дорівнює	10 –eq 10
-ne –cne –ine	Не дорівнює	9 –ne 10
-lt –clt –ilt	Менше	3 –lt 4
-le –cle –ile	Менше або дорівнює	3 –le 4
-gt –cgt –igt	Більше	4 –gt 3
-ge –cge -ige	Більше або дорівнює	4 –ge 3
-contains -ccontains -icintains	Містить	1,2,3 –contains 1
-notcontains -cnotcontains -inotcontains	Не містить	1,2,3 –notcontains 4

Базовий варіант операторів порівняння (-eq, -ne, -lt і т. Д.) За замовчуванням не враховує регістр букв. Якщо оператор починається з букви "c" (-ceq, -cne, -clt і т. д.), то при порівнянні регістр букв буде братися до уваги. Якщо оператор починається з букви "i" (-ieq, -ine, -ilt і т. Д.), то регістр букв при порівнянні не враховується.

Для операторів порівняння справедливо "правило лівої руки" (визначальним є тип лівого операнда). Якщо число порівнюється з рядком, то рядок перетворюється в число і виконується порівняння двох чисел. Якщо лівий операнд є рядком, то правий операнд перетвориться до символьного типу і виконується порівняння двох рядків.

Шаблони з підстановочними символами

У PowerShell підтримуються чотири види групових символів і кілька операторів, які перевіряють рядки на відповідність з шаблонами з підстановочними символами

Оператор	**Опис**	**Приклад (повертається значення $True)**
-like -clike -ilike	Порівняння на збіг з урахуванням підставного символу в тексті	"file.doc" –like "f*.doc"

-notlike -cnotike -inotlike	Порівняння на розбіжність з урахуванням підставного символу в тексті	"file.doc" –notlike "f*.rtf"

Оператори, що працюють з регулярними виразами

Оператор	**Опис**	**Приклад**	**Результат**
-match **–cmatch** **–imatch**	Порівняння на збіг з врахуванням регулярних виразів в правому операнді	"книга" -match "ни" "нос" -match "[к-н]ос"	$True $True
-notmatch **-cnotmatch** **-inotmatch**	Порівняння на збіг з урахуванням регулярних виразів в правому операнді	"книга" -notmatch "^кн"	$False
-replace **-creplace** **-ireplace**	Заміна або видалення символів в рядку - лівому операнде (ці оператори повертають змінену рядок). Якщо в якості правого операнда вказуються через кому два підрядка, то перша з них відповідає фрагменту, який потрібно змінити, а друга - рядку, яка буде вставлена в рядок заміни. Якщо в якості правого операнда вказана одна підстрока, то вона відповідає фрагменту рядка, який буде видалений	"род" -replace "д", "т" "род" -replace "ро"	"рот" "д"

Логічні оператори

Іноді всередині однієї інструкції необхідно перевірити відразу кілька умов. Оператори порівняння можна з'єднувати один з одним за допомогою логічних операторів. При використанні логічного оператора PowerShell перевіряє кожне умова окремо, а потім визначає значення інструкцій цілком, пов'язуючи умови за допомогою логічних операторів.

Логічні оператори в PowerShell

Оператор	**Значення**	**Приклад (повертається значення $True)**
-and	Логічне І	(10 -eq 10) –and (1 –eq 1)
-or	Логічне АБО	(9 -ne 10) –or (3 –eq 4)
-not	Логічне НІ	-not (3 –gt 4)
!	Логічне НІ	!(3 -gt 4)

9. Арифметичні оператори у PowerShell

Арифметичні оператори використовуються для обчислення числових значень. Можна використовувати один або кілька арифметичних операторів для додавання (+), віднімання(-), множення(*) і ділення значень(/), а також для обчислення залишку при операції ділення(%).

Крім того, оператор додавання (+) і оператор множення (*) також використовуються при роботі з рядками, масивами і хеш-таблицями. Оператор додавання об'єднує введені дані. Оператор множення повертає кілька копій вхідних даних. В арифметичному операторі можуть навіть використовуватися об'єкти різних типів. Метод, який використовується для обчислення результату оператора, визначається типом самого лівого об'єкта у виразі.

Оператор	***Опис***	***Приклад***	***Результат***
+	Додає 2 значення	2+4 “aaa” + “bbb” 1, 2, 3 + 4, 5 (масив)	6 “aaabbb” 1, 2, 3, 4, 5
*	Перемножує 2 значення	2*4 “a” * 3 1, 2, 3 * 2	8 “aaa” 1, 2, 3, 1, 2, 3
-	Віднімає одне значення від іншого	5-3	2
/	Ділить одне значення на інше	9/3 7/4	3 1,75
%	Обчислення залишку при операції ділення	7%4	3

10.Масиви у PowerShell, їх використання та дії над масивами.

Масив являє собою структуру даних, яка призначена для зберігання набору елементів. Елементи можуть бути одного і того ж типу або різних типів.
Починаючи з Windows PowerShell 3.0, нульовий елемент або одиночний об'єкт мають деякі властивості масивів.
Щоб створити та ініціалізувати масив, потрібно привласнити кілька значень змінній. Значення, які зберігаються в масиві розділяються комою (,) і відокремлені від імені змінної оператором присвоювання (=).
Наприклад:
$A = 22,5,10,8,12,9,80
Також можна створити та ініціалізувати масив, використовуючи оператор діапазону (..).
Наприклад:
$B = 5..8
В результаті $ B буде містити чотири значення: 5, 6, 7 і 8.
Якщо тип даних не заданий, Windows PowerShell створює масив як масив об'єктів (тип: System.Object []).
Щоб дізнатися тип даних масиву, використовуйте метод GetType (). Наприклад, щоб дізнатися тип даних масиву $ a, виконайте: $a.GetType()
Щоб створити типізований масив, тобто масив, який може містити тільки значення певного типу, наприклад, String [], long [] або int32 [],перед ім'ям змінної масиву в квадратних дужках вкажіть тип. Наприклад, щоб створити масив містить 32-бітові цілі числа з ім'ям $ IA, що містить чотири числа (1500, 2230 3350, і 4000), виконайте: [Int32[]] $ia = 1500,2230,3350,4000

Оператор перевизначення в масив (@)

Оператор перевизначення (@) створює масив, навіть якщо він не містить елементів або містить тільки один об'єкт. Синтаксис оператора масиву виглядає наступним чином:@ (...)
Ви можете використовувати оператор масиву для створення масиву нульового значення або містить один об'єкт. приклад:
$a = @("Один") $a.Count - виведе 1 $b = @() $b.Count - виведе 0

Читання масиву

Звернутися до масиву можна використовуючи ім'я змінної масиву. Щоб відобразити всі елементи в масиві, треба ввести ім'я масиву. наприклад:
$a .До елементів масиву можна звернутися за допомогою індексу, починаючи з 0. Укладаючи число в дужках. Наприклад, щоб відобразити нульовий елемент масиву $ a, треба ввести: $a[0].

Перегляд властивостей масиву

Щоб подивитися властивості і методи масиву, наприклад для отримання довжини масиву (Length) або метод для установки значення елемента в масиві (SetValue), використовуйте параметр InputObject командлет Get-Member.
При передачі масиву по конвеєру в командлет Get-Member, Windows PowerShell відправляє об'єкти по одному і Get-Member повертає тип кожного елемента в масиві. наприклад: $a|Get-Member
При використанні параметра InputObject, Get-Member повертає властивості масиву. Наприклад, наступна команда отримує властивості масиву $ a: Get-Member -InputObject $a

Робота з масивами

Щоб об'єднати два масиви в один масив, використовуйте оператор плюс (+). У наступному прикладі створюється два масиви, і об'єднаються в третій, а потім відображає отриманий об'єднаний масив.
Приклад:
$x = 1,3
$y = 5,9
$z = $x + $y

11. Змінні у PowerShell, типи змінних та їх властивості

Мова PowerShell підтримує всі основні числові типи платформи .Net. При чому немає потреби явно задавати тип числа – система підбере його автоматично

Також передбачена можливість задавати числові літерали за допомогою суфікса множителя (10.1Kb = 10.1*1024, 14Mb = 14*1024*1024), або у шістнадцятковому форматі (0x10 = 16, 0xB = 11).

Символьні літерали задаються за допомогою оголошення тексту в одинарних або двійних лапках, можуть в собі також містити спеціальні знаки і змінні

Змінні, як і все інше в PowerShell нечутливі до регістру. Назва змінної може містити що завгодно, але якщо вона міститиме символи що можуть неоднозначно сприйнятись (наприклад пробіли) то її потрібно помістити в фігурні дужки. Кожна змінна позначається знаком долара ($) за яким йде її назва.

Змінні створюються при першому присвоєнні, наприклад:

$a = $b = 0 # присвоїти обом нуль

$a , $b = 2, 3 # $a буде 2, а $b - 3

Константи

Окрім звичайного присвоєння, змінну можна створити за допомогою командлета New-Variable, що дозволяє передати їй деякі додаткові параметри, наприклад вказати режим лише для читання:

New-Variable -Name pi -Value 3.1415926 -Option ReadOnly

Щоб видалити константу, доведеться передати в Remove-Item ключ -Force.

Змінні середовища

Щоб отримати доступ до змінних середовища, використовують префікс env:, наприклад *$env:path.*