

Цель работы: Знакомство с основными возможностями оболочки командной строки Windows PowerShell 2.0.

Ход работы:

1. Ознакомиться с теоретическим материалом.
2. Выполнить задания.
3. Ответить на контрольные вопросы.

Задание:

1. Ознакомиться с теоретическими сведениями
2. Запустить оболочку PowerShell
3. Увеличить ширину окна оболочки до максимальной, увеличить высоту окна и задать цвет фона и цвет шрифта
4. Вывести содержимое каталога Windows по указанному в таблице 5 формату на экран и в текстовый файл.
5. Вывести в текстовый файл список свойств процесса, возвращаемый командлетом Get-process и на экран – их общее количество.
6. Создать текстовый файл, содержащий список выполняемых процессов, упорядоченный по возрастанию указанного в таблице 6 параметра. Имена параметров процессов указаны в той же таблице.
7. Создать HTML-файл, содержащий список выполняемых процессов, упорядоченный по возрастанию указанного в таблице 6 параметра. Имена параметров процессов указаны в той же таблице.
8. Найти суммарный объем всех графических файлов (bmp, jpg), находящихся в каталоге Windows и всех его подкаталогах.
9. Вывести на экран сведения о ЦП компьютера.
10. Найти максимальное, минимальное и среднее значение времени выполнения командлетов dir и ps
11. Выполнить индивидуальные задания для студентов бригад согласно таблице 7.

Описание выполнения работы

4. Вывести содержимое каталога Windows по указанному в таблице 5 формату на экран и в текстовый файл.

Задачу можно выполнить с помощью следующего скрипта:

```
Get-ChildItem -Path "C:\Windows" -Directory | Where-Object {$_.Name - match "[st]$" } | Sort-Object Name | Select-Object Name, CreationTime, Attributes | Format-Table | Out-File "task4_output.txt"
```

В этом скрипте мы используем командлет Get-ChildItem для получения списка всех подкаталогов в каталоге Windows. Затем мы используем командлет Where-Object для фильтрации подкаталогов, которые имеют имена, оканчивающиеся на s или t. Далее мы используем командлет Sort-Object для сортировки подкаталогов по имени и командлет Select-Object для выбора свойств имени, даты создания и атрибутов.

Результаты выполнения скрипта представлен на рисунке 3.1, а содержимое файла task4_output.txt на рисунке 3.2

Name	CreationTime	Attributes
----	-----	-----
AAct_Tools	29.03.2022 17:06:29	Directory
addins	07.12.2019 17:35:43	Directory, NotContentIndexed
appcompat	07.12.2019 12:14:52	Directory, NotContentIndexed
AppReadiness	07.12.2019 12:14:52	Directory, NotContentIndexed
Boot	07.12.2019 12:14:52	Directory
Containers	07.12.2019 12:14:52	Directory, NotContentIndexed
Cursors	07.12.2019 12:14:52	Directory, NotContentIndexed
diagnostics	07.12.2019 12:14:52	Directory
Downloaded Program Files	07.12.2019 12:14:52	System, Directory, NotContentIndexed
en-US	07.12.2019 17:34:32	Directory, NotContentIndexed
Fonts	07.12.2019 12:14:52	ReadOnly, System, Directory, NotContentIndexed
L2Schemas	07.12.2019 12:14:52	Directory, NotContentIndexed
LiveKernelReports	07.12.2019 12:14:52	Directory, NotContentIndexed
Logs	07.12.2019 12:14:52	Directory, NotContentIndexed
Microsoft.NET	07.12.2019 12:14:52	ReadOnly, Directory, NotContentIndexed
ModemLogs	07.12.2019 12:14:52	Directory, NotContentIndexed
Offline Web Pages	07.12.2019 12:14:52	ReadOnly, Directory, NotContentIndexed
PolicyDefinitions	07.12.2019 12:14:52	Directory, NotContentIndexed
pss	12.04.2022 22:30:13	Directory
RemotePackages	07.12.2019 17:37:33	Directory, NotContentIndexed
Resources	07.12.2019 12:14:52	Directory, NotContentIndexed
schemas	07.12.2019 12:14:52	Directory, NotContentIndexed
ServiceProfiles	06.06.2021 22:42:39	Directory, NotContentIndexed
ShellComponents	07.12.2019 12:14:52	Directory, NotContentIndexed
ShellExperiences	07.12.2019 12:14:52	Directory, NotContentIndexed
SystemApps	07.12.2019 12:14:52	Directory, NotContentIndexed
SystemResources	07.12.2019 12:14:52	Directory
Tasks	07.12.2019 12:14:52	Directory, NotContentIndexed
Vss	07.12.2019 12:14:52	Directory, NotContentIndexed
WaaS	07.12.2019 12:14:52	Directory
WinSxS	07.12.2019 12:03:44	Directory

Рисунок 3.1 – Результат выполнения скрипта для 4 задания

task4_output.txt – Блокнот

Файл Правка Формат Вид Справка

Name	CreationTime	Attributes
----	-----	-----
AAct_Tools	29.03.2022 17:06:29	Directory
addins	07.12.2019 17:35:43	Directory, NotContentIndexed
appcompat	07.12.2019 12:14:52	Directory, NotContentIndexed
AppReadiness	07.12.2019 12:14:52	Directory, NotContentIndexed
Boot	07.12.2019 12:14:52	Directory
Containers	07.12.2019 12:14:52	Directory, NotContentIndexed
Cursors	07.12.2019 12:14:52	Directory, NotContentIndexed
diagnostics	07.12.2019 12:14:52	Directory
Downloaded Program Files	07.12.2019 12:14:52	System, Directory, NotContentIndexed
en-US	07.12.2019 17:34:32	Directory, NotContentIndexed
Fonts	07.12.2019 12:14:52	ReadOnly, System, Directory, NotContentIndexed
L2Schemas	07.12.2019 12:14:52	Directory, NotContentIndexed
LiveKernelReports	07.12.2019 12:14:52	Directory, NotContentIndexed
Logs	07.12.2019 12:14:52	Directory, NotContentIndexed
Microsoft.NET	07.12.2019 12:14:52	ReadOnly, Directory, NotContentIndexed
ModemLogs	07.12.2019 12:14:52	Directory, NotContentIndexed
Offline Web Pages	07.12.2019 12:14:52	ReadOnly, Directory, NotContentIndexed
PolicyDefinitions	07.12.2019 12:14:52	Directory, NotContentIndexed
pss	12.04.2022 22:30:13	Directory
RemotePackages	07.12.2019 17:37:33	Directory, NotContentIndexed
Resources	07.12.2019 12:14:52	Directory, NotContentIndexed
schemas	07.12.2019 12:14:52	Directory, NotContentIndexed
ServiceProfiles	06.06.2021 22:42:39	Directory, NotContentIndexed
ShellComponents	07.12.2019 12:14:52	Directory, NotContentIndexed
ShellExperiences	07.12.2019 12:14:52	Directory, NotContentIndexed
SystemApps	07.12.2019 12:14:52	Directory, NotContentIndexed
SystemResources	07.12.2019 12:14:52	Directory
Tasks	07.12.2019 12:14:52	Directory, NotContentIndexed
Vss	07.12.2019 12:14:52	Directory, NotContentIndexed
WaaS	07.12.2019 12:14:52	Directory
WinSxS	07.12.2019 12:03:44	Directory

Стр 36, столб 1 100% Windows (CRLF) UTF-16 LE

Рисунок 3.2 – Содержимое файла task4_output.txt

5. Вывести в текстовый файл список свойств процесса, возвращаемый командлетом Get-process и на экран – их общее количество

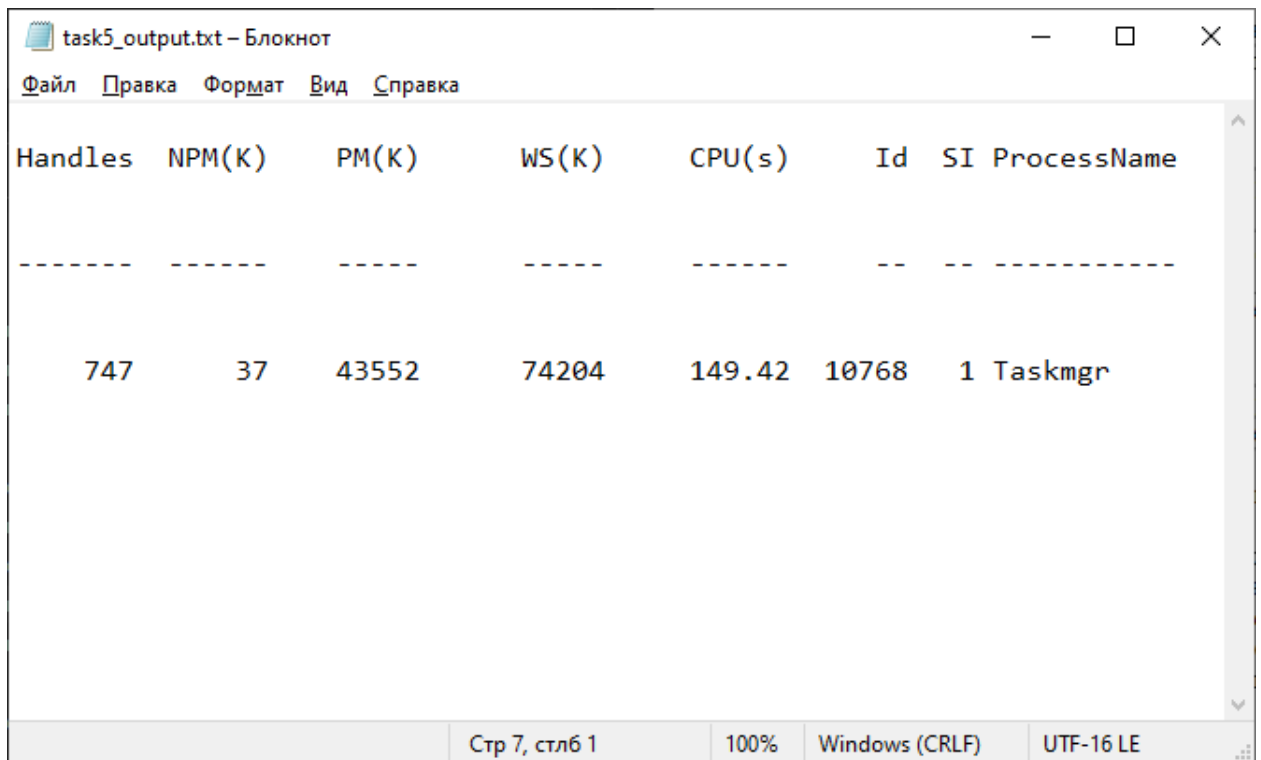
Введем следующую команду для выполнения первой части задания:

Get-Process taskmgr | Out-File "task5_output.txt"

Введем следующую команду для выполнения второй части задания.

(Get-Process).count

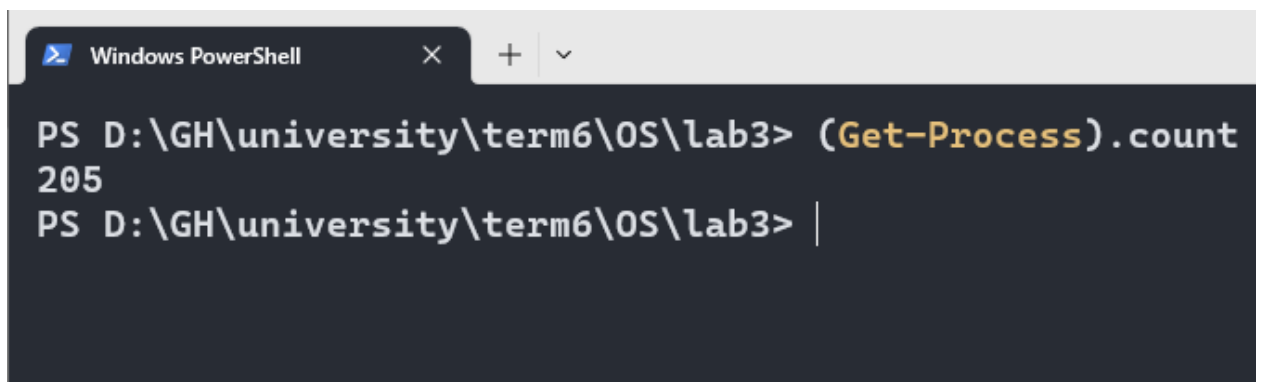
Содержимое файла task5_output.txt представлено на рисунке 3.3, а результат выполнения скрипта для второй части на рисунке 3.4



Handles	NPM(K)	PM(K)	WS(K)	CPU(s)	Id	SI	ProcessName
-----	-----	-----	-----	-----	--	--	-----
747	37	43552	74204	149.42	10768	1	Taskmgr

Стр 7, стлб 1 100% Windows (CRLF) UTF-16 LE

Рисунок 3.3 – Содержимое файла task5_output.txt



```
PS D:\GH\university\term6\OS\lab3> (Get-Process).count
205
PS D:\GH\university\term6\OS\lab3> |
```

Рисунок 3.4 – Результат выполнения скрипта для второй части задания 5

6. Создать текстовый файл, содержащий список выполняемых процессов, упорядоченный по возрастанию указанного в таблице 6 параметра. Имена параметров процессов указаны в той же таблице

Следующий скрипт позволит выполнить данное задание:

```
Get-Process | Where-Object {$_.Id -gt 100} | Select-Object Name,
PriorityClass, ProductVersion, Id | Sort-Object Name | Out-File
"task06_output.txt"
```

Содержимое файла task06_output.txt представлено на рисунке 3.5.

Name	PriorityClass	ProductVersion	Id
----	-----	-----	--
amd64fndrsr			2172
amdow			5912
AMDRSServ			5164
AMDRSSrcExt	Normal	10,01,01,1862	3792
ApplicationFrameHost	Normal	10.0.19041.746	976
atieclxx			2992
atiesrxx			2184
chrome	Normal	111.0.5563.147	9220
chrome	Normal	111.0.5563.147	9404
chrome	Normal	111.0.5563.147	9020
chrome	AboveNormal	111.0.5563.147	8992
chrome	Normal	111.0.5563.147	9000
chrome	Idle	111.0.5563.147	10192
chrome	Idle	111.0.5563.147	10552
chrome	Normal	111.0.5563.147	9700
chrome	Normal	111.0.5563.147	9416
chrome	Normal	111.0.5563.147	9692
chrome	Normal	111.0.5563.147	8756
chrome	Idle	111.0.5563.147	4984
chrome	Normal	111.0.5563.147	5148
chrome	Idle	111.0.5563.147	2740
chrome	Normal	111.0.5563.147	1368
chrome	Idle	111.0.5563.147	1880
chrome	Idle	111.0.5563.147	7584
chrome	Normal	111.0.5563.147	8716
chrome	Normal	111.0.5563.147	6992
chrome	Idle	111.0.5563.147	6228
chrome	Idle	111.0.5563.147	6932
cncmd	Normal	8,01,01,1501	1196
Code	Normal	1.77.0	8180
Code	Normal	1.77.0	7696
Code	Normal	1.77.0	6668
Code	AboveNormal	1.77.0	8552
Code	Normal	1.77.0	10212
Code	Normal	1.77.0	10020
Code	Normal	1.77.0	9824

Рисунок 3.5 – Содержимое файла task06_output.txt

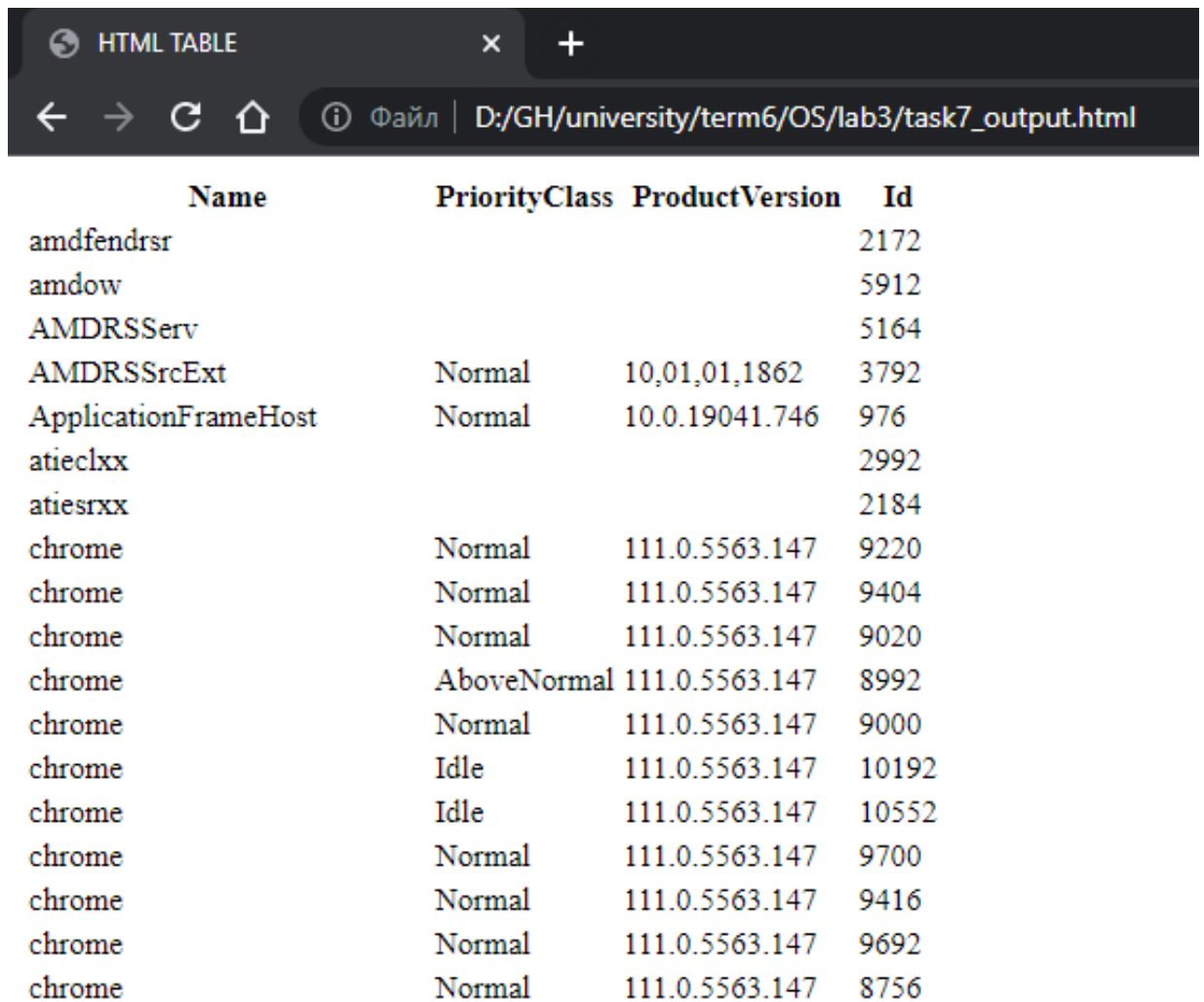
Создадим файл Task1.bat. Напишем код, который будет выполнять данную задачу.

7. Создать HTML-файл, содержащий список выполняемых процессов, упорядоченный по возрастанию указанного в таблице 6 параметра. Имена параметров процессов указаны в той же таблице.

Скрипт, позволяющий выполнить данное задание, имеет вид:

```
Get-Process | Where-Object {$_.Id -gt 100} | Select-Object Name,
PriorityClass, ProductVersion, Id | Sort-Object Name | ConvertTo-Html
> "task7_output.html"
```

Результатом его работы будет содержимое файла task7_output.html. Оно представлено на рисунке 3.7



The screenshot shows a web browser window with the title 'HTML TABLE'. The address bar displays 'D:/GH/university/term6/OS/lab3/task7_output.html'. The table contains the following data:

Name	PriorityClass	ProductVersion	Id
amdfendrsr			2172
amdow			5912
AMDRSServ			5164
AMDRSSrcExt	Normal	10,01,01,1862	3792
ApplicationFrameHost	Normal	10.0.19041.746	976
atieclxx			2992
atiesrxx			2184
chrome	Normal	111.0.5563.147	9220
chrome	Normal	111.0.5563.147	9404
chrome	Normal	111.0.5563.147	9020
chrome	AboveNormal	111.0.5563.147	8992
chrome	Normal	111.0.5563.147	9000
chrome	Idle	111.0.5563.147	10192
chrome	Idle	111.0.5563.147	10552
chrome	Normal	111.0.5563.147	9700
chrome	Normal	111.0.5563.147	9416
chrome	Normal	111.0.5563.147	9692
chrome	Normal	111.0.5563.147	8756

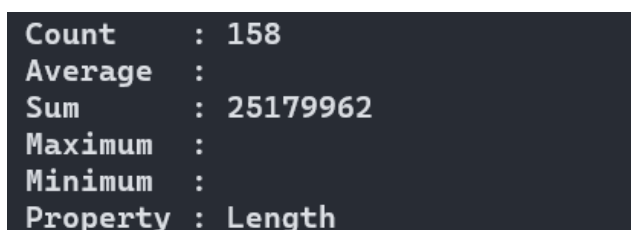
Рисунок 3.7 – Содержимое файла task7_output.html.

8. Найти суммарный объем всех графических файлов (bmp, jpg), находящихся в каталоге Windows и всех его подкаталогах.

Скрипт, позволяющий выполнить данное задание, имеет вид:

```
Get-ChildItem -Path "C:\Windows" -Include "*.jpg", "*.bmp" -Recurse |
Measure-Object -property length -sum
```

Результат его работы представлен на рисунке 3.8.



```
Count      : 158
Average    :
Sum        : 25179962
Maximum    :
Minimum    :
Property   : Length
```

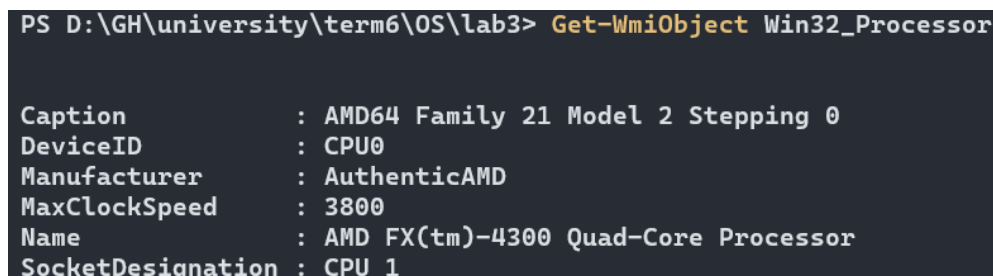
Рисунок 3.8 – Результат работы скрипта для выполнения задания 8

9. Вывести на экран сведения о ЦП компьютера.

Следующий скрипт позволит выполнить задание:

```
Get-WmiObject Win32_Processor
```

Результат его работы представлен на рисунке 3.9



```
PS D:\GH\university\term6\OS\lab3> Get-WmiObject Win32_Processor

Caption           : AMD64 Family 21 Model 2 Stepping 0
DeviceID          : CPU0
Manufacturer      : AuthenticAMD
MaxClockSpeed     : 3800
Name              : AMD FX(tm)-4300 Quad-Core Processor
SocketDesignation : CPU 1
```

Рисунок 3.9 – Информация о ЦП компьютера

10. Найти максимальное, минимальное и среднее значение времени выполнения командлетов dir и ps

Для нахождения времени работы команды в миллисекундах необходимо использовать команду:

```
(Measure-Command { dir }).TotalMilliseconds
```

Для выполнения задания необходимо сделать несколько измерений, для сохранения результатов которых воспользуемся массивом, который создается следующим образом:

```
$c = New-Object System.Collections.ArrayList
```

Добавление элемента происходит при помощи функции Add.

Для вывода минимального, максимального и среднего значения массива воспользуемся функцией:

```
measure -Maximum -Minimum -Average
```

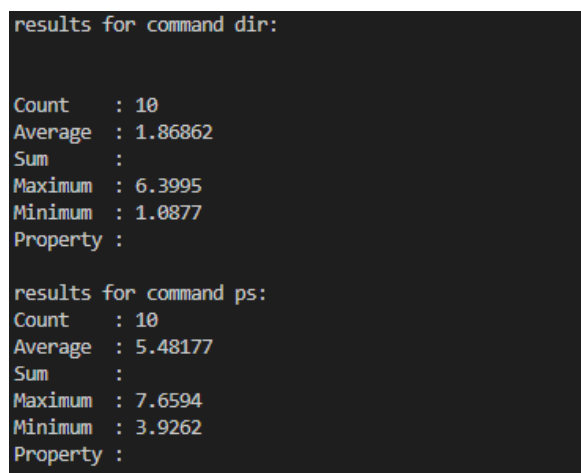
Таким образом, получаем следующий код:

```
$c = New-Object System.Collections.ArrayList
for ($i = 1; $i -le 10; $i++) {
    $c.Add((Measure-Command { Get-ChildItem }).TotalMilliseconds) >
    $null
}
Write-Output "results for command dir: "
$c | Measure-Object -Maximum -Minimum -Average
```

```
$c = New-Object System.Collections.ArrayList
for ($i = 1; $i -le 10; $i++) {
    $c.Add((Measure-Command { Get-Process }).TotalMilliseconds) >
    $null
}
Write-Output "results for command ps: "
$c | Measure-Object -Maximum -Minimum -Average
```

Результат его выполнения отображен на рисунке 3.10

Вывод: по результатам проведенного эксперимента на текущей конфигурации компьютера под управлением ОС Windows команда `dir` работает быстрее команды `ps`.



```
results for command dir:
Count      : 10
Average     : 1.86862
Sum         :
Maximum     : 6.3995
Minimum     : 1.0877
Property    :

results for command ps:
Count      : 10
Average     : 5.48177
Sum         :
Maximum     : 7.6594
Minimum     : 3.9262
Property    :
```

Рисунок 3.10 – Время работы командлетов `dir` и `ps`

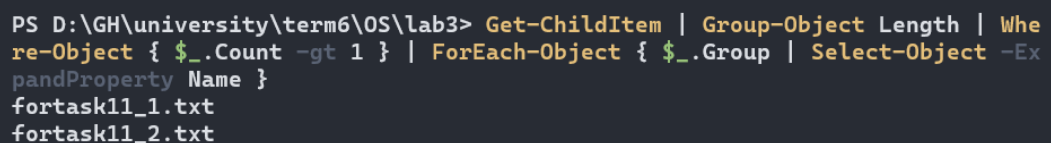
11. Выполнить индивидуальные задания:

11.1 Проверить наличие в текущем каталоге файлов одинакового размера. Если такие файлы есть – вывести их имена.

Задание можно выполнить, используя этот скрипт:

```
Get-ChildItem | Group-Object Length | Where-Object { $_.Count -gt 1 }
| ForEach-Object { $_.Group | Select-Object -ExpandProperty Name }
```

Результат его выполнения отображен на рисунке 3.11.



```
PS D:\GH\university\term6\OS\lab3> Get-ChildItem | Group-Object Length | Where-Object { $_.Count -gt 1 } | ForEach-Object { $_.Group | Select-Object -ExpandProperty Name }
fortask11_1.txt
fortask11_2.txt
```

Рисунок 3.11 – Список файлов одинакового размера текущего каталога

Контрольные вопросы

1. Типы команд PowerShell (PS).

В оболочке PowerShell поддерживаются команды четырех типов: командлеты, функции, сценарии и внешние исполняемые файлы. Первый тип – так называемые командлеты (cmdlet). Этот термин используется пока только внутри PowerShell. Командлет – аналог внутренней команды интерпретатора командной строки – представляет собой класс .NET, порожденный от базового класса Cmdlet; разрабатываются командлеты с помощью пакета PowerShell Software Developers Kit (SDK). Единый базовый класс Cmdlet гарантирует совместимый синтаксис всех командлетов, а также автоматизирует анализ параметров командной строки и описание синтаксиса командлетов для встроенной справки. Командлеты рассматриваются в данной работе.

Данный тип команд компилируется в динамическую библиотеку (DLL) и подгружается к процессу PowerShell во время запуска оболочки (то есть сами по себе командлеты не могут быть запущены как приложения, но в них содержатся исполняемые объекты). Командлеты – это аналог внутренних команд традиционных оболочек. Следующий тип команд – функции. Функция – это блок кода на языке PowerShell, имеющий название и находящийся в памяти до завершения текущего сеанса командной оболочки. Функции, как и командлеты, поддерживают именованные параметры. Анализ синтаксиса функции производится один раз при ее объявлении. Сценарий – это блок кода на языке PowerShell, хранящийся во внешнем файле с расширением ps1. Анализ синтаксиса сценария производится при каждом его запуске. Последний тип команд – внешние исполняемые файлы, которые выполняются обычным образом операционной системой.

2. Имена и структура командлетов.

В PowerShell аналогом внутренних команд являются командлеты. Командлеты могут быть очень простыми или очень сложными, но каждый из

них разрабатывается для решения одной, узкой задачи. Работа с командлетами становится по-настоящему эффективной при использовании их композиции (конвейеризации объектов между командлетами).

Команды Windows PowerShell следуют определенным правилам именования: Команды Windows PowerShell состоят из глагола и существительного (всегда в единственном числе), разделенных тире. Глагол задает определенное действие, а существительное определяет объект, над которым это действие будет совершено. Команды записываются на английском языке. Пример: `Get-Help` вызывает интерактивную справку по синтаксису Windows PowerShell. Перед параметрами ставится символ «-». Например: `Get-Help - Detailed`. В Windows PowerShell также включены псевдонимы многих известных команд. Это упрощает знакомство и использование Windows PowerShell. Пример: команды `help` (классический стиль Windows) и `man` (классический стиль Unix) работают так же, как и `Get-Help`. Например, `Get-Process` (получить информацию о процессе), `Stop-Service` (остановить службу), `Clear-Host` (очистить экран консоли) и т.д. Чтобы просмотреть список командлетов, доступных в ходе текущего сеанса, нужно выполнить командлет `Get-Command`. По умолчанию командлет `Get-Command` выводит сведения в трех столбцах: `CommandType`, `Name` и `Definition`. При этом в столбце `Definition` отображается синтаксис командлетов (многоточие (...) в столбце синтаксиса указывает на то, что данные обрезаны). Замечание. Косые черты (/ и \) вместе с параметрами в оболочке Windows PowerShell не используются. В общем случае синтаксис командлетов имеет следующую структуру: `имя_командлета -параметр1 -параметр2 аргумент1 аргумент2`. Здесь `параметр1` – параметр (переключатель), не имеющий значения; `параметр2` – имя параметра, имеющего значение `аргумент1`; `аргумент2` – параметр, не имеющий имени. Например, командлет `Get-Process` имеет параметр `Name`, который определяет имя процесса, информацию о котором нужно вывести. Имя этого параметра указывать необязательно. Таким образом, для получения сведений о процессе

Far можно ввести либо команду `Get-Process -Name Far`, либо команду `Get-Process Far`.

3. Псевдонимы команд.

Механизм псевдонимов, реализованный в оболочке PowerShell, дает возможность пользователям выполнять команды по их альтернативным именам (например, вместо команды `Get-Childitem` можно пользоваться псевдонимом `dir`). В PowerShell заранее определено много псевдонимов, можно также добавлять собственные псевдонимы в систему. Псевдонимы в PowerShell делятся на два типа. Первый тип предназначен для совместимости имен с разными интерфейсами. Псевдонимы этого типа позволяют пользователям, имеющим опыт работы с другими оболочками (`Cmd.exe` или Unix-оболочки), использовать знакомые им имена команд для выполнения аналогичных операций в PowerShell, что упрощает освоение новой оболочки, позволяя не тратить усилий на запоминание новых команд PowerShell. Например, пользователь хочет очистить экран. Если у него есть опыт работы с `Cmd.exe`, то он, естественно, попыбует выполнить команду `cls`. PowerShell при этом выполнит командлет `Clear-Host`, для которого `cls` является псевдонимом и который выполняет требуемое действие – очистку экрана. Для пользователей `Cmd.exe` в PowerShell определены псевдонимы `cd`, `cls`, `copy`, `del`, `dir`, `echo`, `erase`, `move`, `popd`, `pushd`, `ren`, `rmdir`, `sort`, `type`; для пользователей Unix – псевдонимы `cat`, `chdir`, `clear`, `diff`, `h`, `history`, `kill`, `lp`, `ls`, `mount`, `ps`, `pwd`, `r`, `rm`, `sleep`, `tee`, `write`. Узнать, какой именно командлет скрывается за знакомым псевдонимом, можно с помощью командлета `Get-Alias`.

Псевдонимы второго типа (стандартные псевдонимы) в PowerShell предназначены для быстрого ввода команд. Такие псевдонимы образуются из имен командлетов, которым они соответствуют. Например, глагол `Get` сокращается до `g`, глагол `Set` сокращается до `s`, существительное `Location` сокращается до `l` и т.д. Таким образом, для командлету `SetLocation` соответствует псевдоним `sl`, а командлету `Get-Location` – псевдоним `gl`.

Просмотреть список всех псевдонимов, объявленных в системе, можно с помощью командлета `Get-Alias` без параметров. Определить собственный псевдоним можно с помощью командлета `Set-Alias`.

4. Просмотр структуры объектов.

Для анализа структуры объекта, возвращаемого определенной командой, проще всего направить этот объект по конвейеру на командлет `Get-Member` (псевдоним `gm`), например:

```
PS C:\> Get-Process | Get-Member
```

Мы увидим имя .NET-класса, экземпляры которого возвращаются в ходе работы исследуемого командлета, а также полный список элементов объекта (в частности, интересующее нас свойство `Responding`, определяющего "зависшие" процессы). При этом на экран выводится очень много элементов, просматривать их неудобно. Командлет `Get-Member` позволяет перечислить только те элементы объекта, которые являются его свойствами. Для этого используется параметр `MemberType` со значением `Properties`:

```
PS C:\> Get-Process | Get-Member -MemberType Property
```

Процессам ОС соответствуют объекты, имеющие очень много свойств, на экран же при работе командлета `Get-Process` выводятся лишь несколько из них (способы отображения объектов различных типов задаются конфигурационными файлами в формате XML, находящимися в каталоге, где установлен файл `powershell.exe`).

5. Фильтрация объектов в конвейере. Блок сценария.

В PowerShell поддерживается возможность фильтрации объектов в конвейере, т.е. удаление из конвейера объектов, не удовлетворяющих определенному условию. Данную функциональность обеспечивает командлет `Where-Object`, позволяющий проверить каждый объект, находящийся в конвейере, и передать его дальше по конвейеру, только если

объект удовлетворяет условиям проверки. Например, для вывода информации о «зависших» процессах (объекты, возвращаемые командлетом Get-Process, у которых свойство Responding равно False) можно использовать следующий конвейер:

```
Get-Process | Where-Object {-not $_.Responding}
```

Другой пример – оставим в конвейере только те процессы, у которых значение идентификатора (свойство Id) больше 1000:

```
Get-Process | Where-Object {$_.Id -gt 1000} 67
```

В блоках сценариев командлета Where-Object для обращения к текущему объекту конвейера и извлечения нужных свойств этого объекта используется специальная переменная `$_`, которая создается оболочкой PowerShell автоматически. Данная переменная используется и в других командлетах, производящих обработку элементов конвейера. Условие проверки в Where-Object задается в виде блока сценария – одной или нескольких команд PowerShell, заключенных в фигурные скобки `{}`. Результатом выполнения данного блока сценария должно быть значение логического типа: True (истина) или False (ложь). Как можно понять из примеров, в блоке сценария используются специальные операторы сравнения.

Замечание. В PowerShell для операторов сравнения не используются обычные символы `>` или `<` так как в командной строке они обычно означают перенаправление ввода/вывода.

Оператор	Значение	Пример (возвращается значение True)
-eq	равно	10 -eq 10
-ne	не равно	9 -ne 10
-lt	меньше	3 -lt 4
-le	меньше или равно	3 -le 4
-gt	больше	4 -gt 3
-ge	больше или равно	4 -ge 3
-like	сравнение на совпадение с учетом подстановочного знака в тексте	"file.doc" -like "f*.doc"
-notlike	сравнение на несовпадение с учетом подстановочного знака в тексте	"file.doc" -notlike "f*.rtf"
-contains	содержит	1,2,3 -contains 1
-notcontains	не содержит	1,2,3 -notcontains 4

Операторы сравнения можно соединять друг с другом с помощью логических операторов (см. таблице 2).

Оператор	Значение	Пример (возвращается значение True)
-and	логическое И	(10 -eq 10) -and (1 -eq 1)
-or	логическое ИЛИ	(9 -ne 10) -or (3 -eq 4)
-not	логическое НЕ	-not (3 -gt 4)
!	логическое НЕ	!(3 -gt 4)

6. Какую информацию выводит команда Get-Help * ?

Get-Help * перечисляет все команды Windows PowerShell.

7. Командлеты для форматирования выводимой информации.

В традиционных оболочках команды и утилиты сами форматируют выводимые данные. Некоторые команды (например, dir в интерпретаторе Cmd.exe) позволяют настраивать формат вывода с помощью специальных параметров. В оболочке PowerShell вывод форматируют только четыре специальных командлета Format (таблица 4). Это упрощает изучение, так как не нужно запоминать средства и параметры форматирования для других команд (остальные командлеты вывод не форматируют).

Командлет	Описание
Format-Table	Форматирует вывод команды в виде таблицы, столбцы которой содержат свойства объекта (также могут быть добавлены вычисляемые столбцы). Поддерживается возможность группировки выводимых данных
Format-List	Вывод форматируется как список свойств, в котором каждое свойство отображается на новой строке. Поддерживается возможность группировки выводимых данных
Format-Custom	Для форматирования вывода используется пользовательское представление (view)
Format-Wide	Форматирует объекты в виде широкой таблицы, в которой отображается только одно свойство каждого объекта

Как уже отмечалось, если ни один из командлетов Format явно не указан, то используется модуль форматирования по умолчанию, который определяется по типу отображаемых данных.

Для изменения формата выводимых данных нужно направить их по конвейеру соответствующему командлету Format. Например, следующая команда выведет список служб с помощью командлета Format-List.

При использовании формата списка выводится больше сведений о каждой службе, чем в формате таблицы (вместо трех столбцов данных о каждой службе в формате списка выводятся девять строк данных). Однако это вовсе не означает, что командлет Format-List извлекает дополнительные сведения о службах. Эти данные содержатся в объектах, возвращаемых командлетом Get-Service, однако командлет FormatTable, используемый по умолчанию, отбрасывает их, потому что не может вывести на экран более трех столбцов. При форматировании вывода с помощью командлетов FormatList и Format-Table можно указывать имена свойства объекта, которые должны быть отображены (напомним, что просмотреть список свойств, имеющихся у объекта, позволяет рассмотренный ранее командлет Get-Member).

Вывести все имеющиеся у объектов свойства можно с помощью параметра *.

8. Перенаправление выводимой информации.

В оболочке PowerShell имеются несколько командлетов, с помощью которых можно управлять выводом данных. Эти командлеты начинаются со слова Out, их список можно получить с помощью командлета:

```
PS C:\> Get-Command out-* | Format-Table Name
```

По умолчанию выводимая информация передается командлету OutDefault, который, в свою очередь, делегирует всю работу по выводу строк на экран командлету Out-Host. Для понимания данного механизма нужно учитывать, что архитектура PowerShell подразумевает различие между собственно ядром оболочки (интерпретатором команд) и главным приложением (host), которое использует это ядро. В принципе, в качестве главного может выступать любое приложение, в котором реализован ряд специальных интерфейсов, позволяющих корректно интерпретировать получаемую от PowerShell информацию. В нашем случае главным приложением является консольное окно, в котором мы работаем с оболочкой, и командлет Out-Host передает выводимую информацию в это консольное окно. Параметр Paging командлета Out-Host, подобно команде more интерпретатора Cmd.exe, позволяет организовать постраничный вывод информации, например: `Get-Help Get-Process -Full | Out-Host -Paging`.

9. Управляющие инструкции PS.

А) Инструкция If ... ElseIf ... Else

В общем случае синтаксис инструкции If имеет вид

```
If (условие1) {блок_кода1}  
[ElseIf (условие2)] {блок_кода2}]  
[Else {блок_кода3}].
```

При выполнении инструкции If проверяется истинность условного выражения условие1. Если условие1 имеет значение \$True, то выполняется блок_кода1, после чего выполнение инструкции if завершается. Если условие1 имеет значение \$False, проверяется истинность условного выражения условие2. Если условие2 имеет значение \$True, то выполняется

блок_кода2 и выполнение инструкции if завершается. Если и условие1, и условие2 имеют значение \$False, то выполняется блок_кода3 и выполнение инструкции if завершается.

Б) Циклы While и Do ... While

Самый простой из циклов PS – цикл While, в котором команды выполняются до тех пор, пока проверяемое условие имеет значение \$True. Инструкция While имеет следующий синтаксис: While (условие) {блок_команд} Цикл Do ... While похож на цикл While, однако условие в нем проверяется не до блока команд, а после: Do {блок_команд} While (условие).

В) Цикл For Обычно цикл For применяется для прохождения по массиву и выполнения определенных действий с каждым из его элементов. Синтаксис инструкции For:

For (инициация; условие; повторение) {блок_команд}

Г) Цикл ForEach Инструкция ForEach позволяет последовательно перебирать элементы коллекций. Самый простой тип коллекции – массив. Особенность цикла ForEach состоит в том, что его синтаксис и выполнение зависят от того, где расположена инструкция ForEach: вне конвейера команд или внутри конвейера. Инструкция ForEach вне конвейера команд: В этом случае синтаксис цикла ForEach имеет вид:

ForEach (\$элемент in \$коллекция) {блок_команд}

При выполнении цикла ForEach автоматически создается переменная \$элемент. Перед каждой итерацией в цикле этой переменной присваивается значение очередного элемента в коллекции. В разделе блок_команд содержатся команды, выполняемые на каждом элементе коллекции. Инструкция ForEach может также использоваться совместно с командлетами, возвращающими коллекции элементов.

10. Назначение регулярных выражений.

Регулярные выражения (или сокращенно “регэкспы” (regex, regular expressions)) обладают огромной мощностью, и способны сильно упростить

жизнь системного администратора или программиста. В PowerShell регулярные выражения легко доступны, удобны в использовании и максимально функциональны. PowerShell использует реализацию регулярных выражений .NET.

Регулярные выражения - это специальный мини-язык, служащий для разбора (parsing) текстовых данных. С его помощью можно разделять строки на компоненты, выбирать нужные части строк для дальнейшей обработки, производить замены и т. д.

Знакомство с регулярными выражениями начнем с более простой технологии, служащей подобным целям – с подстановочных символов. Наверняка вы не раз выполняли команду `dir`, указывая ей в качестве аргумента маску файла, например `*.exe`. В данном случае звёздочка означает “любое количество любых символов”. Аналогично можно использовать и знак вопроса, он будет означать “один любой символ”, то есть `dir ??.exe` выведет все файлы с расширением `.exe` и именем из двух 71 символов.

В PowerShell можно применять и еще одну конструкцию – группы символов. Так например `[a-f]` будет означать “один любой символ от а до f, то есть (a,b,c,d,e,f)”, а `[smw]` любую из трех букв (s, m или w). Таким образом команда `get-childitem [smw]???.exe` выведет файлы с расширением `.exe`, у которых имя состоит из трех букв, и первая буква либо s, либо m, либо w.

Для начала изучения мы будем использовать оператор PowerShell - `match`, который позволяет сравнивать текст слева от него, с регулярным выражением справа. В случае если текст подпадает под регулярное выражение, оператор выдаёт `True`, иначе – `False`.

```
PS C:\> "PowerShell" -match "Power"
```

```
True
```

При сравнении с регулярным выражением ищется лишь вхождение строки, полное совпадение текста необязательно (разумеется, это можно изменить). То есть достаточно, чтобы регулярное выражение встречалось в тексте.

```
PS C:\> "Shell" -match "Power"
```

False

```
PS C:\> "PowerShell" -match "rsh"
```

True Еще одна тонкость: оператор `-match` по умолчанию не чувствителен к регистру символов (как и другие текстовые операторы в PowerShell), если же нужна чувствительность к регистру, используется `-cmatch`.

11. Сохранение данных в текстовом файле и html-файле.

Для преобразования данных в формат html служит командлет `Convertto-html`. Параметр `Property` определяет свойства объектов, включаемые в выходной документ. Например, для получения списка выполняемых процессов в формате html, включающего имя процесса и затраченное время CPU и записи результата в файл `processes.html` можно использовать команду

```
Get-Process | Convertto-html -Property Name, CPU > Processes.htm
```

Для просмотра содержимого файла можно использовать командлет `Invoke-Item` «имя документа» Например `Invoke-Item "processes.htm"`

12. Получение справочной информации в PS.

Вместо `help` или `man` в Windows PowerShell можно также использовать команду `Get-Help`. Ее синтаксис описан ниже:

`Get-Help` выводит на экран справку об использовании справки.

`Get-Help *` перечисляет все команды Windows PowerShell.

`Get-Help` команда выводит справку по соответствующей команде.

`Get-Help` команда `-Detailed` выводит подробную справку с примерами команды.

Использование команды `help` для получения подробных сведений о команде `help`: `Get-Help Get-Help -Detailed`.

Команда Get-Help позволяет просматривать справочную информацию не только о разных командлетах, но и о синтаксисе языка PowerShell, о псевдонимах и т. д. Например, чтобы прочитать справочную информацию об использовании массивов в PowerShell, нужно выполнить следующую команду: `Get-Help about_array`.

Командлет Get-Help выводит содержимое раздела справки на экран сразу целиком. Функции `man` и `help` позволяют справочную информацию выводить построчно (аналогично команде MORE интерпретатора Cmd.exe), например: `man about_array`.

13. Как создать массив в PS?

Для создания и инициализации массива достаточно присвоить значения его элементам. Значения, добавляемые в массив, разделяются запятыми и отделяются от имени массива символом присваивания. Например, следующая команда создаст массив \$a из трех элементов:

```
PS C:\> $a=1,5,7
```

Можно создать и инициализировать массив, используя оператор диапазона (..). Например, команда `PS C:\> $b=10..15` создает и инициализирует массив \$b, содержащий 6 значений 10, 11, 12, 13, 14 и 15.

Для создания массива может использоваться операция ввода значений его элементов из текстового файла:

```
PS C:\> $f = Get-Content c:\data\numb.txt -TotalCount 25
```

```
PS C:\> $f.length 25
```

В приведенном примере результат выполнения командлета GetContent присваивается массиву \$f. Необязательный параметр `-TotalCount` ограничивает количество прочитанных элементов величиной 25. Свойство объекта массив `length` имеет значение, равное количеству элементов массива, в примере оно равно 25 (предполагается, что в текстовом файле numb.txt по крайней мере 25 строк).

14. Как объединить два массива?

Можно объединить два массива, например \$b и \$c в один с помощью операции конкатенации +. Например: PS C:\> \$d=\$b+\$c.

15. Как увеличить размер созданного в PS массива?

Имеется способ увеличения первоначально определенной длины массива. Для этого можно воспользоваться оператором конкатенации + или +=.

```
PS C:\>$a+=5,6
```

16. Как ввести данные в массив?

Следующая команда создаст массив \$a из трех элементов:

```
PS C:\> $a=1,5,7
```

Можно инициализировать массив, используя оператор диапазона (..). Например, команда PS C:\> \$b=10..15 создает и инициализирует массив \$b, содержащий 6 значений 10, 11, 12, 13, 14 и 15.

Для создания массива может использоваться операция ввода значений его элементов из текстового файла:

```
PS C:\> $f = Get-Content c:\data\numb.txt -TotalCount 25
```

```
PS C:\>$f.length 25
```

17. Использование командлета Out-Null.

Командлет Out-Null служит для поглощения любых своих входных данных. Это может пригодиться для подавления вывода на экран ненужных сведений, полученных в качестве побочного эффекта выполнения какой-либо команды.

18. Оператор PowerShell –match.

Для начала изучения мы будем использовать оператор PowerShell -match, который позволяет сравнивать текст слева от него, с регулярным выражением справа. В случае если текст подпадает под регулярное выражение, оператор выдаёт True, иначе – False.

```
PS C:\> "PowerShell" -match "Power"
True
```

При сравнении с регулярным выражением ищется лишь вхождение строки, полное совпадение текста необязательно (разумеется, это можно изменить). То есть достаточно, чтобы регулярное выражение встречалось в тексте.

```
PS C:\> "Shell" -match "Power"
False
```

```
PS C:\> "PowerShell" -match "rsh"
True
```

Еще одна тонкость: оператор -match по умолчанию не чувствителен к регистру символов (как и другие текстовые операторы в PowerShell), если же нужна чувствительность к регистру, используется -cmatch:

```
PS C:\> "PowerShell" -cmatch "rsh"
False
```

19. Использование символа ^ в командлетах.

"Крышка" в качестве первого символа группы символов означает именно отрицание. То есть на месте группы может присутствовать любой символ кроме перечисленных в ней.

Для того чтобы включить отрицание в символьных группах (\d, \w, \s), не обязательно заключать их в квадратные скобки, достаточно перевести их в верхний регистр.

^ как символ отрицания используется лишь в начале группы символов, а вне группы – этот символ является уже якорем.

20. Использование символа \$ в командлетах.

\$ (знак доллара) - обозначает конец строки.

21. Количественные модификаторы (квантификаторы).

В регулярных выражениях существует специальная конструкция – «количественные модификаторы» (квантификаторы). Эти модификаторы приписываются к любой группе справа, и определяют количество вхождений этой группы.

Как и в случае с символьными группами, для особенно популярных значений количественных модификаторов, есть короткие псевдонимы:

+ (плюс), эквивалентен {1,} то есть, "одно или больше вхождений"

* (звездочка), то же самое что и {0,} или на русском языке – "любое количество вхождений, в том числе и 0"

? (вопросительный знак), равен {0,1} – "либо одно вхождение, либо полное отсутствие вхождений".

В регулярных выражениях, количественные модификаторы сами по себе использоваться не могут. Для них обязателен символ или символьная группа, которые и будут определять их смысл.

22. Использование групп захвата.

Как следует из названия, группы можно использовать для группировки. К группам захвата, как и к символам и символьным группам, можно применять количественные модификаторы.

Например, следующее выражение означает «Первая буква в строке – S, затем одна или больше групп, состоящих из «знака – (минус) и любого количества цифр за ним» до конца строки»:

```
PS C:\> "S-1-5-21-1964843605-2840444903-4043112481" -match "^S(-\d+)$"
```

True

23. Командлеты для измерения свойств объектов.

Для измерения времени выполнения командлетов PS служит командлет Measure-Command. В качестве примера можно рассмотреть получение времени выполнения командлета dir, выполнив команду (Measure-Command {dir}).TotalSeconds.

Для получения статистических данных используют командлет Measure-Object. Для числовых массивов с его помощью можно получить максимальное, минимальное, среднее значение элементов массива и их сумму. Если имеется инициализированный массив ms, для указанной цели используется командлет \$ms | measure-object -maximum -minimum -average -sum.