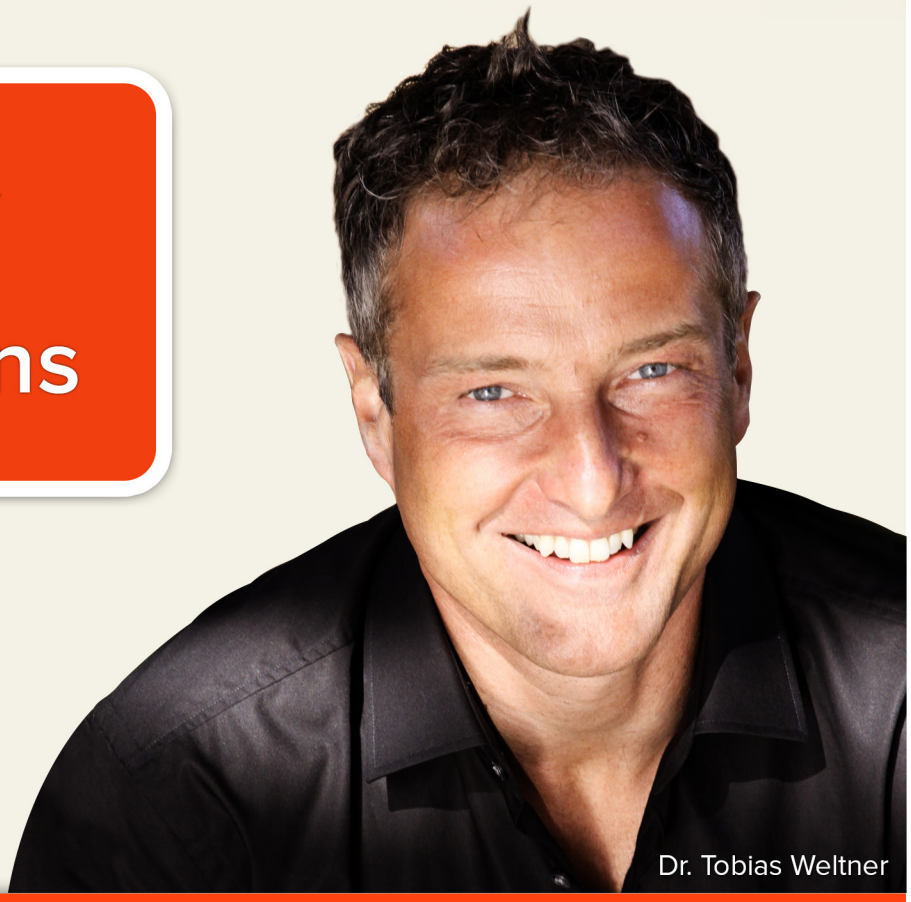
PowerTips MONTHLY

Part of the PowerShell.com reference library, brought to you by **Dr. Tobias Weltner**

Volume 6 November 2013

This Month's Topic:

Regular Expressions



Dr. Tobias Weltner

Table of Contents

1. PowerShell Support for Regular Expressions

PowerShell supports regular expressions in many operators such as -split, -replace, and -match. In addition, regular expressions can be used in conjunction with the RegEx type that provides a number of very powerful RegEx methods to find and replace text.

2. Regular Expression Pattern Reference

Regular expressions are patterns that describe what you are looking for.

You typically compose a regular expression with three ingredients: placeholders, quantifiers, and anchors (or you simply navigate to Google and search for the regular expression pattern you need; since regular expressions are mostly platform-independent, you can grab any one you find and try it for yourself).

Regular expressions provide a number of placeholders that you can use to specifically describe what you want:

Placeholder	Description
.	Any character except newline (Equivalent: [^\n])
[^abc]	All characters except the ones specified
[^a-z]	All characters except those in the region specified
[abc]	One of the characters
[a-z]	One of the characters in the region
\a	Bell (ASCII 7)
\c	Any character allowed in XML names
\cA-\cZ	Control+A to Control+Z, ASCII 1 to ASCII 26
\d	Any number (Equivalent: [0-9])
\D	Any non-number
\e	Escape (ASCII 27)
\f	Form Feed, (ASCII 12)
\n	Line break
\r	Carriage return
\s	Any whitespace (space, tab, new line)
\S	Any non-whitespace
\t	tab
\w	Letter, number or underline
\W	Non-letter, number, or underline

In addition, you can use quantifiers. They tell RegEx how often your placeholder occurs. Without a quantifier, each placeholder always represents exactly one instance.

Quantifier	Description
*	Any (no occurrence, once, many times)
?	No occurrence or one occurrence
{n,}	At least n occurrences
{n,m}	At least n occurrences, maximum m occurrences
{n}	Exactly n occurrences
+	One or many occurrences

Each quantifier by default looks for the longest match it can find (greedy). If you want to get the shortest possible match, add another “?”.

Finally, you can use anchors to tie the pattern to some position in your text. Anchors can be plain text, so “KB\d” would find any number that has a “KB” prefix. You can also use these popular anchors:

$	End of text
^	Start of text
\b	Word boundary
\B	No word boundary
\G	After last match (no overlaps)

Regular expressions are, by default, case sensitive. When you use PowerShell operators, you control case sensitivity by picking the appropriate operator (-replace is case-insensitive, whereas -creplace is case-sensitive).

When you work with raw RegEx types and objects, prepend your expression with “(?i)” to make it case-insensitive.

You will now find many practical and working examples. Use them, or try and look up the ingredients of the regular expressions in the tables above.

3. Converting Semicolons and Tabs to Commas

Occasionally, you may have to convert "CSV" content to real CSV. Depending on regional settings, CSV may use commas, semicolons or tabs as delimiters. This will convert commas and tabs to semicolons:

```
PS> 'Unit1,Unit2,Unit3' -replace '[,\t]', ';'
Unit1;Unit2;Unit3
```

To convert an entire file, like windowsupdate.log (which is by default tab separated), try this:

```
$newContent = foreach ($line in (Get-Content $env:windir\WindowsUpdate.log -ReadCount 0))
{
  $line -replace '\t', ','
}

$header = Write-Output Date Time Code1 Code2 Type Topic Response DetailedError Code3 Code4 ID Code5
Code6 Origin InstallResult Action ActionResponse Remark

$newContent | ConvertFrom-Csv -Header $header | Out-GridView
```

As you see, your entire raw tab-separated log file becomes now a manageable object-oriented grid. Note that $header contains any text you want. This is a list of column headers that you can supply when your raw data input has no own column headers.

Now it is easy to get a detailed report on the latest Windows updates installed:

```
$newContent = foreach ($line in (Get-Content $env:windir\WindowsUpdate.log -ReadCount 0))
{
  $line -replace '\t', ','
}

$header = Write-Output Date Time Code1 Code2 Type Topic Response DetailedError Code3 Code4 ID Code5 Code6 Origin InstallResult
Action ActionResponse Remark

$newContent |
   ConvertFrom-Csv -Header $header |
   Where-Object { $_.Action } |
   Select-Object -Property Date, Time, Origin, Action, ActionResponse, InstallResult, Remark |
   Out-GridView
```

Author Bio

Tobias Weltner is a long-term Microsoft PowerShell MVP, located in Germany. Weltner offers entry-level and advanced PowerShell classes throughout Europe, targeting mid- to large-sized enterprises. He just organized the first German PowerShell Community conference which was a great success and will be repeated next year (more on www.pscommunity.de).. His latest 950-page "PowerShell 3.0 Workshop" was recently released by Microsoft Press.

To find out more about public and in-house training, get in touch with him at tobias.weltner@email.de.

4. Identifying and Extracting Information

The -match operator can both identify and also extract wanted information from raw text.

This is an example of identifying a pattern. This line checks to see whether the given pattern is part of the text:

```
$text = 'PC678 had a problem'
$pattern = 'PC(\d{3})'

$text -match $pattern
True
```

Whenever the -match is positive ($true), PowerShell also extracts the information that matched, and puts it into the $matches variable:

```
PS> $matches

Name                           Value
----                           -----
1                              678
0                              PC678
```

$matches is a hash table actually. Key "0" always holds the match for the entire pattern:

```
PS> $matches[0]
PC678
```

If there are braces in your pattern, then there is additional matches information, one for each brace-pair.

5. Comparison Operators Turn to Filters When Applied to Arrays

Most comparison operators (including -match) work differently when applied to arrays (more than one value).

When applied to arrays, comparison operators no longer return $true or $false. Instead, they become filters. They filter out all array elements that do not match.

This would select only text that contains "IPv4":

```
PS> (ipconfig) -match 'IPv4'
   IPv4 Address. . . . . . . . . . . : 172.20.10.3
```

And this would select only log file lines with “successfully installed” in them:

```
PS> (Get-Content C:\Windows\WindowsUpdate.log) -match 'successfully installed'
```

So basically, while -match will only find the first match in each line, it can be used to find multiple matches in log files.

First, use -match to identify those lines in a text that contain the pattern you are after. Second, apply -match again on each of these lines to find the actual information.

This will extract recent updates from the log file windowsupdate.log inside the Windows folder:

```
$patternProduct = 'update: (.*)'
$patternKB = 'KB(\d{5,9})'

(Get-Content C:\Windows\WindowsUpdate.log) -match 'successfully installed' |
  ForEach-Object {
    $result = 1 | Select-Object -Property Date, KB, Product

    if ($_ -match $patternProduct)
    {
      $result.Product = $matches[1]
    }
    if ($_ -match $patternKB)
    {
      $result.KB = $matches[1]
    }
    $result.Date = [DateTime] ($_.SubString(0,10) + ' ' + $_.SubString(11, 8))

    $result
  } | Out-GridView -Title 'Recently installed updates'
```

6. Extracting IP Information from ipconfig.exe

You can apply regular expressions to any text, even text returned by native console commands like ipconfig.exe. To extract your IPv4 information, use a regular expression that looks for anything that seems to be an IP address. While there are more sophisticated regular expressions, for this task it is sufficient to look for any four numbers with a length of 1 to 3 that have dots in between them.

Also make sure your regular expression won’t identify any IPv4 address by providing an anchor such as “IPv4” or “Subnet”. Between the anchor and the actual IP address, add “.*?” which represents “anything”:

```
$pattern = '.*?((\d{1,3}\.){3}\d{1,3})'
$info    = ipconfig
$ip      = $info -match "IPv4$pattern" | ForEach-Object { if ($_ -match $pattern) { $matches[1] }}
$subnet  = $info -match "Subnet$pattern" | ForEach-Object { if ($_ -match $pattern) { $matches[1] }}
$gateway = $info -match "Gateway$pattern" | ForEach-Object { if ($_ -match $pattern) { $matches[1]
}}

"IP: $ip Subnet: $subnet Gateway: $gateway"
```

Technical Editor Bio

Aleksandar Nikolic, Microsoft MVP for Windows PowerShell, a frequent speaker at the conferences (Microsoft Sinergija, PowerShell Deep Dive, NYC Techstravaganza, KulenDayz, PowerShell Summit) and the cofounder and editor of the PowerShell Magazine (http://powershellmagazine.com). He is also available for one-on-one online PowerShell trainings. You can find him on Twitter: https://twitter.com/alexandair

When you run Get-WmiHelpLocation, it opens the web page in your default browser that documents the WMI class you specified, and also returns the URL:

```
PS> Get-WmiHelpLocation Win32_Share
http://msdn.microsoft.com/en-us/library/aa394435(VS.85).aspx
```

7. Removing Multiple White Spaces

Removing multiple white spaces from text is easy in PowerShell. Simply use -replace operator and look for whitespaces ("\s") that occur one or more time ("+"), then replace them all with just one whitespace:

```
PS> '[     Man,     it works!     ]' -replace '\s+', ' '
[ Man, it works! ]
```

8. Turning Fixed Width Columns into CSV

Replacing multiple whitespaces is a key when you need to turn fixed-width formatted text into CSV format.

Qprocess.exe, for example, is a tool that returns detailed information about running processes. This information uses fixed width columns:

```
PS> qprocess.exe
 USERNAME              SESSIONNAME         ID    PID  IMAGE
>tobias                console              1   3312  taskhost.exe
>tobias                console              1   3792  dwm.exe
>tobias                console              1   1172  explorer.exe
```

To parse those, replace two or more whitespace with one comma:

```
PS> (qprocess) -replace '\s{2,}', ','
 USERNAME,SESSIONNAME,ID,PID,IMAGE
>tobias,console,1,3312,taskhost.exe
>tobias,console,1,3792,dwm.exe
>tobias,console,1,1172,explorer.exe
```

Now you can feed the standard CSV format into ConvertFrom-Csv, and get back real objects:

```
PS> (qprocess) -replace '\s{2,}', ',' | ConvertFrom-CSV | Format-Table

USERNAME        SESSIONNAME     ID              PID             IMAGE
--------        -----------     --              ---             -----
>tobias         console         1               3312            taskhost.exe
>tobias         console         1               3792            dwm.exe
>tobias         console         1               1172            explorer.exe
>tobias         console         1               3828            bootcamp.exe
>tobias         console         1               448             msseces.exe
>tobias         console         1               3876            igfxtray.exe
(...)
```

9. Using Regular Expressions with Get-ChildItem

When you use Dir (alias: Get-ChildItem) to list folder contents, you can use simple wildcards but they do not give you much control.

A much more powerful approach is to use regular expressions. Since Get-ChildItem does not support regular expressions, you can use Where-Object to filter the results returned by Dir. This line will get you any file with a number in its filename, ignoring numbers in the file extension:

```
PS> dir $home -Recurse | Where-Object {$_.Name -match '\d.*?\.'} | Select-Object -ExpandProperty
Name
map1.hta
PSConfig64.EXE
03-09-2013 02-07-29.pdf
Boardingpass_X32274.pdf
mirrorfile1.txt
myAT&T v8.9.lnk
kanu1.jpg
(...)
```

10. Normalizing Paths

Sometimes, paths are not well formatted. They may contain combinations of backslashes and/or forward slashes. The -replace operator can normalize these paths because you can create a regular expression that matches both slashes and replaces them with something else:

```
PS> $Path = 'C:\this/is/a\path.txt'

PS> $Path -split '[\/]' -join '\'
C:\this\is\a\path.txt

PS> $Path -split '[\/]' -join '\\'
C:\\this\\is\\a\\path.txt
```

Use square brackets to specify all the characters that you want to replace:

```
PS> 'I replace commas, and also periods.' -replace '[.,]','STOP'
I replace commasSTOP and also periodsSTOP
```

11. Replacing Multiple Instances

Regular expressions have a lot of predefined place holders such as "\W" (which represents any non-word character. You can use this to split or replace on any instance:

```
PS> $text = "Some sample text. This    text contains   [[many]] nonword chars!!!"


PS> $text -replace '\W', '*'
Some*sample*text**This****text*contains*****many***nonword*chars***

PS> $text -replace '\W+', '*'
Some*sample*text*This*text*contains*many*nonword*chars*

PS> $text -replace '\W+', '$0'
Some sample text. This    text contains   [[many]] nonword chars!!!
```

```
PS> $text -replace '(\W)+', '$1'

Some sample text This text contains[many nonword chars!

PS> $text -replace '(\W)(\1)+', '$1'

Some sample text. This text contains [many] nonword chars!
```

12. Finding and Extracting Text

With regular expressions, you can easily extract matching text. Have a look:

```
$text = 'The problem was discussed in KB552356. Mail feedback to tobias @powershell.com'
$pattern = 'KB\d{4,6}'
if ($text -match $pattern)
{
      $KB = $matches[0]
      "The KB was $KB"
}
else
{
      'There was no KB number in the text.'
}
```

13. Splitting Without Losing Anything

When you split text, the split character by default is consumed. Thus, it is missing in the result. Here is a line that wants to get a file extension part of a path:

```
PS> $profile -split '\.'
C:\Users\Tobias\Documents\WindowsPowerShell\Microsoft
PowerShellISE_profile
ps1
```

It works, but the dot is gone. By using a clever trick, you can include the split character:

```
PS> $profile -split '(?=\.)'
C:\Users\Tobias\Documents\WindowsPowerShell\Microsoft
.PowerShellISE_profile
.ps1

PS> ($profile -split '(?=\.)')[-1]
.ps1
```

This is called "look ahead", and with "?=" you are instructing RegEx to split right after the pattern you specify (without taking out anything).

This is a very powerful technique. It lets you do much more sophisticated things. Here is an example that extracts parameters from an argument string:

```
PS> $commandline = '-Path c:\windows -Filter *.exe'

PS> $commandline -split '(?=-)(?<=\s)'
-Path c:\windows
-Filter *.exe
```

In addition to "?=", you here also use a "look behind" ("?<="). So this regular expression looks ahead to find a "-", and then also looks behind to find any whitespace. Without this, it would split before any "-".

14. Extracting Words

Assume you have an expression with multiple words. Words are not separated by spaces but instead by upper case letters. Using look ahead and look behind expressions, you can easily extract the words:

```
PS> 'GetHostByName' -csplit '(?<=[a-z])(?=[A-Z])'
Get
Host
By
Name
```

15. Scraping Information from HTML Websites

Regular expressions can extract valuable information from raw HTML. Here's sample code that downloads HTML from any website. If you do not have direct Internet access and use a proxy and/or credentials, you will want to add more parameters to Invoke-WebRequest.

```
$result = Invoke-WebRequest -Uri www.powershell.com
$html = $result.Content
```

We could now, for example, scrape every instance of a hyperlink from that website:

```
$patternLink = '(i?)<a href="(?<link>.*)".*>(?<text>.*)</a>'
$MatchLink = [RegEx]$patternLink

$MatchLink.Matches($html) |
  ForEach-Object { $_.Value } |
  Out-GridView
```

You can now even access the matches found by subexpressions. Subexpressions are braces in your pattern. In the sample pattern, this is the href location and the actual text displayed:

```
$MatchLink.Matches($html) |
  ForEach-Object {
          $rv = 1 | Select-Object -Property Title, Uri
          $rv.Title = $_.Groups[3].Value
          $rv.Uri = $_.Groups[2].Value
          $rv
    } |
  Out-GridView
```

It works!

You may argue that the result returned by Invoke-WebRequest is already clever enough to provide you this information:

```
$result = Invoke-WebRequest -Uri www.powershell.com
$result.Links |
  Select-Object -Property outerText, href |
  Out-GridView
```

That's true (and truly cool), but with regular expressions you are free to scrape whatever information you want, not just the types of information that Invoke-WebRequest has for you.

16. Creating Pairs of Two

If you'd have to process a long list of encoded information, let's say a list of hexadecimal values, how would you split the list into pairs of two? Here is a way:

```
PS> 'this gets splitted in pairs of two' -split '(?<=\G.{2})(?=.)'
th
is
 g
et
s
sp
li
tt
ed
(...)
```

17. Splitting Hex Values

This would be more specific and split only hex values:

```
PS> '00AA1CFFAB1034' -split '(?<=\G[0-9a-f]{2})(?=.)'
00
AA
1C
FF
AB
10
(...)
```

Now it's easy to reformat MAC addresses as well:

```
PS> '00AA1CFFAB1034' -split '(?<=\G[0-9a-f]{2})(?=.)' -join ':'
00:AA:1C:FF:AB:10:34
```

And you can also take hex dumps and convert them to decimals:

```
$dump = '00AA1CFFAB1034'
foreach ($hex in ($dump -split '(?<=\G[0-9a-f]{2})(?=.)'))
{
  [Convert]::ToByte($hex, 16)
}
```

18. Matching Stars

Asterisk serves as a wildcard, so how would you check for the presence of an asterisk? It's much harder than you might think:

```
PS> 'Test*' -eq '*'
False

PS> 'Test*' -like '**'
True

PS> 'Test' -like '**'
True
```

None of those work. Regular expressions do, because here you can escape (invalidate) any wildcard using the backslash ("\"):

```
PS> 'Test*' -match '\*'
True

PS> 'Test' -match '\*'
False
```

There is an even easier way that won't require regular expressions at all:

```
PS> 'Test*'.Contains('*')
True
```

19. Escaping Regular Expressions

Many PowerShell operators support regular expressions: -split, -replace, and -match, for example. Because these operators expect regular expressions, your code fails if you accidentally use regex keywords like in the following command:

```
PS> 'c:\test\subfolder\file.txt' -split '\'
parsing "\" - Illegal \ at end of pattern.
At line:1 char:1
```

To make the code work, you need to escape special characters with "\":

```
PS> 'c:\test\subfolder\file.txt' -split '\\'
c:
test
subfolder
file.txt

PS> ('c:\test\subfolder\file' -split '\\')[-1]
File.txt

PS> ('c:\test\subfolder\file' -split '\\')[0]
c:
```

```
PS> (‘c:\test\subfolder\file’ -split ‘\\’)[0,1]
c:
test

PS> (‘c:\test\subfolder\file’ -split ‘\\’)[0,1] -join ‘\’
c:\test
```

There is automatic escaping available so you don’t have to remember escaping yourself:

```
PS> [Regex]::Escape(‘\’)
\\

PS> [Regex]::Escape(‘c:\test\subfolder\file.txt’)
c:\\test\\subfolder\\file\.txt
```

20. Replacing Text

The -replace operator supports regular expressions. If you wanted to replace the last octet in an IP address by a new number, a simple replace wouldn’t let you do this. A regular expression replace turns this into a piece of cake:

```
PS> ‘192.168.1.200’ -replace ‘\d{1,3}$’,’201’
192.168.1.201
```

In this example, the RegEx pattern represents a 1-3 digit number at the end of a text (“$”). It is replaced with a new value of “201.”

21. Replacing Text with References to Old Values

Maybe you do not want to replace a value with a completely new static value. Instead, you might want to use part of the old match in your new value:

```
PS> $list = ‘server1’, ‘server2’, ‘server12’
PS> $pattern = ‘server(\d{1,3})’
PS> $list -replace $pattern, ‘VM_$1’

VM_1
VM_2
VM_12
```

As you can see, the server names in your list were replaced with “VM”, but the server number remained the same.

The new value used “$1” (which is not a PowerShell variable in this context, so the text must be single quoted!). This refers to the match found in the first subexpression (first braces) which happens to be the number.

Likewise, “$0” refers to the entire match:

```
PS> $list -replace $pattern, ‘VM_$1 (was $0 before)’
VM_1 (was server1 before)
VM_2 (was server2 before)
VM_12 (was server12 before)
```

22. Replacing Text with Calculated Values

If you want to replace text with a new text, and the new text should be a dynamic value based on the original value, the built-in Replace() method found in the RegEx type can help. It accepts a script block that does the calculation of the replacement value.

Let's assume you wanted to take an IP address and increment the last octet. Here is how:

```
[Regex]::Replace('192.168.1.1', '\d{1,3}$', {param($OldValue) [Int]$OldValue.Value + 1})
```

The script block receives one parameter called $OldValue which is the match. This object has a property called "Value" that delivers the actual old value. You have to make sure you convert this old value into the appropriate type (like a number) to do calculations. Without this, it would be treated as string, and PowerShell would simply add a "1" to the string.

23. Finding Multiple RegEx Matches with Select-String

Select-String is the only built-in way of finding multiple matches. It is not very fast, but it gets the job done.

```
$patternEmail = '\b[A-ZO-9._%+-]+@[A-ZO-9.-]+\.[A-Z]{2,4}\b'
$text = "My email is tobias.weltner@email.de and also tobias@powershell.de"

$text | Select-String -AllMatches $patternEmail |
  Select-Object -ExpandProperty Matches |
  Select-Object -ExpandProperty Value
```

If you do not feel like remembering these lines all the time, create a simple filter function like this:

```
filter Matches($pattern) {
      $_ | Select-String -AllMatches $pattern |
           Select-Object -ExpandProperty Matches |
           Select-Object -ExpandProperty Value
}
```

Now you can easily find multiple matches:

```
PS> $text | Matches $patternEmail
tobias.weltner@email.de
tobias@powershell.de
```

24. Finding Multiple RegEx Patterns Fast

To find all occurrences of a pattern in a text, and to find them very fast, use the .NET RegEx type. Here is a sample:

```
$text = 'multiple emails like tobias.weltner@email.de and tobias@powershell.de in a string'
$patternEmail = '(?i)\b([A-ZO-9._%-]+@[A-ZO-9.-]+\.[A-Z]{2,4})\b'

$result = [RegEx]::Matches($text, $patternEmail)
$result.Value
```

This will work only in PowerShell 3.0 and above. To make the code work in any version, try this:

```
$text = 'multiple emails like tobias.weltner@email.de and tobias@powershell.de in a string'
$patternEmail = '(?i)\b([A-ZO-9._%-]+@[A-ZO-9.-]+\.[A-Z]{2,4})\b'

$result = [RegEx]::Matches($text, $patternEmail)
Foreach ($match in $result) { $match.Value }
```

Note the statement "(?i)" in the regular expression pattern description. The RegEx object by default works case-sensitive. To ignore case, use this control statement.

25. Eliminating Duplicate Words

Let's assume you want to eliminate duplicate words in a text. Here is how you can do this:

```
$text = 'this text text contains duplicate words words following each other'
$pattern = '\b(\w+)(\s+\1){1,}\b'
$text -replace $pattern, '$1'
```

Basically, the pattern looks for words ("\b(\w+)") followed by one or more whitespaces ("(\s+)") plus the previous match ("\1") that occurs one or more times ("{1,}"). This is then replaced by the first match ("$1").

26. Extracting Email Addresses

You can create a specialized RegEx pattern matcher by converting a regular expression into the RegEx type. This generates an object that will always match the pattern you submitted.

This creates a pattern matcher for email addresses:

```
$MatchEmail = [RegEx]'(?i)\b[A-ZO-9._%+-]+@[A-ZO-9.-]+\.[A-Z]{2,4}\b'
```

To find one or more email addresses in any text, you now simply use its Matches() method:

```
$text = "My email is tobias.weltner@email.de and also tobias@powershell.de"
$MatchEmail.Matches($text).Value
```

This works in PowerShell 3.0 and better. To make it work in all versions, try this:

```
$MatchEmail = [RegEx]'(?i)\b[A-ZO-9._%+-]+@[A-ZO-9.-]+\.[A-Z]{2,4}\b'
$text = "My email is tobias.weltner@email.de and also tobias@powershell.de"
foreach ($match in $MatchEmail.Matches($text)) { $match.Value }
```