WINDOWS POWERSHELL CHEATSHEET.S

WINDOWS POWERSHELL is the successor of the WINDOWS CMD language, which itself has its roots in the MS-DOS Bat language. All recent versions of Windows offer PowerShell (PS). PS may be seen as Microsoft's answer to the shells common in Unix/Linux (such as CSH, BASH, *etc.*). Its name implies that Microsoft sees the shell as powerful, which it arguably is.

In these notes some important PS commands are listed and PowerShell's most notable feature, the object pipeline, is discussed. From the outset it is important to note that, in contrast to Linux/Unix, *Windows PowerShell is completely case-insensitive.* Nevertheless these notes follow the custom of the MS documentation by using camel case (mixed upper- and lowercase within one word). Directories in a file path are separated by forward slashes (/) in Unix, while in MS-DOS they are separated by backward slashes (\). PowerShell allows both types, but in output prefers the MS-DOS convention.

The monospace text snippets below are valid PS and may be copied, pasted, and executed in a PowerShell- or a PowerShell_ISE-session. This is why the notes form a "Cheatsheet". As is common for cheatsheets, there is hardly any explanation, the examples speak for themselves. It must be stressed here that many of the basic PS commands are not at all orthogonal, so that many variant pipelines can lead to the same effect. Often an example is one out of a multitude of possibilities accomplishing the same task.

The last two sections are about search in and traversal of the Windows Registry by means of PowerShell.

Contents

1 Execution of PowerShell and scripts within PowerShell
2 About PowerShell ISE
3 Cmdlets
4 PSdrives
5 Pipelines
6 Useful aliases
7 Example of a pipelined command
8 More examples of pipelines
9 The cmdlet Where-Object
10 The cmdlet Select-Object
11 The cmdlet ForEach-Object
12 Select-String
13 Create an item
14 Rename an item
15 Typecasting, strings, constants, and arrays
16 The formatting of strings
17 Hash tables
18 A practical example: a table of WiFi passwords
19 Comparison
20 Builtin classes
21 Random numbers
22 Errors
23 Functions
24 Switch
25 Scripts
26 Traversing the Windows Registry
27 Registry lookup

1Execution of PowerShell and scripts within PowerShell

A few preliminary points:

1. There are different routes to open a PowerShell session in a MS-Windows environment:
    - Right click the start button in the task bar: two PowerShell commands appear, one for ordinary users and one for administrators. The second one prompts for an admin password.
    - If you want to start PS in a specific directory, direct Windows (File) Explorer to this directory and then left click the file tab. You see "Open Windows PowerShell", hover your mouse over it and you get the choice of either opening PS in user mode or admin mode. In both cases PS starts in the given directory. Alternatively, you may enter in the address bar of Explorer: "powershell.exe" or "powershell ise".
2. To protect ordinary users against malignant PS scripts, the execution of PowerShell scripts (including profile scripts) must be allowed explicitly. This is done once and for all by issuing the PS command:

```
Set-ExecutionPolicy RemoteSigned -Scope CurrentUser
```

4. Execution of a script requires that the scriptname is prefixed by its path. If the script is in the present folder use `.\script.ps1`, where `script` is the name of the script.
5. Allow execution of a script downloaded from the internet by:

```
unblock-file .\script.ps1
```

   where `script` is the name of the downloaded script (which is in the current directory).

7. Start a new PowerShell session in administrator mode:

```
Start-Process PowerShell -Verb runAs
```

   You will be prompted for the admin password.

2About PowerShell ISE

3Cmdlets

The internal commands of PowerShell are called "cmdlets". A cmdlet name is of the form "verb-noun", where "verb" is one out of a fixed set of verbs. All cmdlets return one or more objects into a pipeline (see below). At the end of a pipeline some selected properties of the current object(s) are written to the screen together with their values. For instance, the one stage pipeline:

```
PS C:\Users\myname> Get-ChildItem
```

returns a collection of objects that belong to the different files in the present folder. Because the command `Get-ChildItem` is the end stage of a (one-stage) pipeline, some selected properties (namely Mode, LastWriteTime, Length, Name) together with their values are written to screen, one line per file.

The names of members (methods and properties) of an object can be obtained by piping the output of a cmdlet to `Get-Member` (see below). Any cmdlet parameter (aka flag) can be truncated to the extent that it is still unique. A few examples of cmdlets and their parameters:

```
    Get-Help                    # Gets help about a cmdlet.
                                # Example: get-help
    Get-PSdrive                 # List available PSdrives, such
                                # as c:, env:, hklm:, hkcu:, alias:, etc.
    Get-ChildItem               # In the Registry: children are subkeys
                                # of the current key.
    Get-ChildItem               # In the File System: children are
subfolders and filenames
                                # of the current folder.
    Get-ChildItem -recurse      # Lists recursively all children
                                # of current PSdrive, Folder, or Registry
key.
    Get-ChildItem -rec -force   # Include hidden folders
                                # (flag -hidden searches hidden directories
only)
    (Get-ChildItem).name        # List names ('name' is an object property)
                                # of files and directories in current
folder.
    Get-ChildItem  -name        # Equivalent to (Get-ChildItem).name
                                #
    (Get-ChildItem).count       # Number of entries in the collection of
objects returned by Get-ChildItem
                                # (count is a property of such a
collection).
```

Flags (parameters) can be used to change the order of the arguments of a cmdlet. Example:

```
    Move-Item *.txt subdirectory
```

moves all `.txt` files to the subfolder `subdirectory`. If, for some reason, the order of the arguments must be changed, we cannot issue from `subdirectory` the following command to move all `.txt` files back (one folder up):

```
Move-Item ..\ *.txt   # Error!
```

This gives the error message:

```
Move-Item : Cannot move item because the item  is in use.
```

The flag `-Destination` remedies this and

```
Move-Item  -Destination ..\ *.txt
```

moves without complaint all `.txt` files one level up, as does the flagless command in different order:

```
Move-Item  *.txt  ..\
```

4PSdrives

A PSDrive is a collection of entities that are grouped such that they may be accessed as a filesystem drive. The grouping is performed by a "PSprovider". By default a PS session has access to about a dozen PSdrives among which `c:, env:, alias:, HKLM:`. Here `c:` is the usual Windows c-drive; `env:` is the space of Windows environmental variables; `alias:` is the collection of cmdlet aliases; `HKLM:` refers to a hive in the Registry.

Normally one enters a PS session in the home folder (home directory) of the user. To switch a PS session to another PSdrive or folder and get the children of the new location, proceed as follows:
Switch to env:

```
Set-Location env:                 # Prompt character becomes `Env:\>`
                                  # (environment variables)
Env:\> Get-Childitem              # Get all environment variables
                                  #
Env:\> Get-Childitem userprofile  # Get environment variable
`userprofile`
                                  # (returns with: USERPROFILE
C:\Users\user_name)
```

Switch to alias:

```
Env:\> Set-Location alias:        # Prompt character becomes Alias:\>
Alias:\> Get-Childitem            # All children (i.e., all aliases)
```

Back to default drive:

```
Alias:\> Set-Location C:\          # Prompt character becomes C:\>
                                   #
C:\> Set-Location $env:userprofile # Use environment variable
`userprofile` to
                                   # switch to C:\Users\user_name (home
folder of user).
C:\Users\user_name>$alias:ls       # Get what alias 'ls' stands for
```

```
                                        # (namely Get-ChildItem)
```

Here `$env:` and `$alias:` (note the $ prefixing the names) refer to the PSdrives `env:` and `alias:`, respectively. Thus, the prefixing by the variables `$env:` and `$alias:` gives access to the respective PSdrives without need to actually change to these drives.

To contents

5Pipelines

IMPORTANT: *Cmdlets pass **objects** through pipelines, **not character streams** as in Unix.*

The pipeline character is `|` (ASCII 124). It must be followed by a command that can handle the output passed through the pipeline; usually this is a cmdlet. Example of a pipeline consisting of three stages:

```
Get-ChildItem *.txt | Where-Object length -lt 1000 | Sort-Object length
```

This returns a list of the names and properties of files with extension `.txt` in the current folder. Shown are the `.txt` files that have a size of less than 1000 bytes. The list is sorted on file size.

Provided you have a file named `flop.txt` in your current folder, `Get-ChildItem flop.txt` will return an object. By the pipe

```
Get-ChildItem flop.txt | Get-Member
```

a full list is obtained of the 50 members of the object associated with `flop.txt`. In object-oriented languages "members" are sometimes called *attributes* of which there are commonly two kinds: *methods* (functions) and *properties* of the object. One of the properties of the object `flop.txt` is `Length` and another one is named `Name`.

Members are selected by a dot, as is usual in object-oriented languages. Thus, one can write:

```
(Get-ChildItem flop.txt).Name  # -> flop.txt
```

One of the properties of any file is `LastWriteTime`. This property can be set to the present date and time without affecting the content of the file (cf. `touch` in Unix):

```
(Get-ChildItem file.txt).LastWriteTime = Get-Date  # "touch" file.txt
```

The object created by a cmdlet depends in general on the cmdlet's parameters (flags). For example, the adding of the flag `-Name`

```
(Get-ChildItem flop.txt -Name).Name  # -> null
```

gives an empty result. Inspection by `Get-Member` shows that indeed the object created by `Get-ChildItem flop.txt -Name` does not posses a member named `Name`.

The object that is passed through a pipeline is referred to by the *automatic variable* `$_`, which *only may be used inside a script block*. (A script block is a collection of statements enclosed in curly brackets.) Accordingly, a member named "`member_name`" of the object passed is referred to as `$_.member_name`.

The cmdlet `Rename-Item` may be used to change file names and file extensions. The old file name can be entered through a pipeline. The new name follows the parameter `-Newname`

```
    "flop.txt" | Rename-Item  -Newname flap.txt  # We now have a file named flap.txt in the folder
```

Invoking the automatic variable, gives the following trivial renaming:

```
    Get-ChildItem flap.txt | Rename-Item -New {$_.name} # Renames flap.txt to flap.txt
```

However, if the piped object `$_` does not have a member named `Name`, an error occurs:

```
   Get-ChildItem flap.txt -Name | Rename-Item -New {$_.Name} # Error: parameter $_.Name is null
```

When a cmdlet enters more than one object into the pipeline (as `Get-ChildItem *` does), these objects are first stored in a temporary buffer. After the buffer is filled, the cmdlet in the second stage performs its task by looping over the buffer and reading the objects one by one. When the second stage is not the last of the pipeline, this cmdlet enters its output objects into yet another temporary buffer that serves as the input for the third stage. Properties of an object that drops off the end of the pipeline are written to screen by a screen writing method of the object; usually this method writes only a subset of the properties of the object. The concept of intermediate buffering is very important in understanding the actions that occur within pipelines.

For example, in the following statement `Get-ChildItem` fills a temporary buffer with objects that all have the property `Basename` (which is also an object). The cmdlet `Select-Object` selects the basename properties and enters them into yet another temporary buffer, which is passed to `Sort-Object`, which sorts the elements of the buffer and writes the names of the basename objects, i.e., `Sort-Object` writes the file names without file extension.

```
    Get-ChildItem | Select-Object Basename | Sort-Object * -Descending
```

The result on the screen is a list of the basenames (names before the first dot) of all files in the current folder. The list is in descending alphabetic order.

To contents

6Useful aliases

Many cmdlets—which by definition have a rather long name—have one or more short aliases. Often an alias is DOS- or Unix-like.

```
    ac = Add-Content                  # Example: ac -value 'The End' -path flop.txt
                                      # (appends `The End` to flop.txt)
    cat = gc = type = Get-Content     # Get the content of a file;
                                      # returns an array with one line per element
    cd = sl = Set-Location            # Change folder, Registry key, or PSdrive.
                                      # Example: cd env:, cd HKLM:
```

```
    cls = clear = Clear-Host          # Clears console
                                      #
    del = erase= rm = Remove-Item    # Remove files, registry keys, etc.
                                      #
    dir = ls = gci = Get-Childitem    # List children in current PSdrive,
folder, Registry key
                                      #
    echo = write = Write-Output       # String to output array. Array is
sent to console, into
                                      # pipeline, or redirected/appended to
file
    foreach = % = Foreach-Object      # Only in pipeline: for each object
crossing the pipeline
                                      # Do not confuse with language
construct of the same name
    ft = Format-Table                 # Example: ls *.jpg |ft directory,
length, name -AutoSize  -Wrap
                                      # (-Autosize and -Wrap are flags of
ft)
    fl = Format-List                  # Example: ls env:Path |fl
                                      # (gives line wrapped output of
environment variable "Path")
    gal = Get-Alias                   # "Get-Alias -definition cmdlet",
gives aliases of cmdlet
                                      # "Get-Alias [-name] alias", gives
name of cmdlet called by alias
    gcm = Get-Command                 # Get all commands (cmdlets,
functions, and aliases).
                                      # gcm -CommandType Alias -> all
aliases
    gm = Get-Member                   # Example: ls flop.txt | gm
                                      # (all members of object flop.txt)
    gp  = Get-ItemProperty            # In file system: gp * gives same
output as ls *
                                      # In Registry: value entries (names
and values)
    gpv  = Get-ItemPropertyValue      # In filesystem: get prop's of files.
Ex: gpv *.txt basename (names of .txt files)
                                      # Ex: In
HKCU:\SOFTWARE\Microsoft\Accessibility: gpv -name cursorsize (returns
number)
    gv = Get-Variable                 # Get names and values of all session
variables
                                      #
    ni = New-Item                     # Create new file, directory, symbolic
link,
                                      # registry key, or registry entry

    ps = gps = Get-Process            # List running processes.
                                      #
```

```
    pwd = gl = Get-Location          # Current directory (folder) or
Registry key
                                     #
    ren = rni = Rename-Item          # Examples: ren report.doc report.txt
                                     # and: ls report.doc | ren -newname
report.txt
    rmdir = rm = ri = Remove-Item    # Remove directories, files, registry
keys, etc.
                                     #
    rv = Remove-Variable             # Remove variable (note: name without
$ prefix, while
                                     # variable names must begin with $)
    sc = Set-Content                 # Example: sc -value 'The End' -path
flop.txt
                                     # (writes `The End` to flop.txt;
overwrites existing content)
    select = Select-Object           # Select specified properties of piped
object
                                     # Example: ps |select Processname  -
first 10
    sleep = Start-Sleep              # Sleep -sec 1 (sleep 1 second)
                                     #
    sls    = Select-String           # Example: sls foo.txt -patt '^\S' (a
regular expression
                                     # giving all lines that do not start
with blank, tab, or EOL)
    where  = ? = Where-Object        # Only in pipelines.
                                     # Example: ls -recurse |? name -like
'*Pict*'
    ~ = $env:userprofile             # Example: cd ~ (change folder to home
folder of user).
```

To contents

7Example of a pipelined command

The following four stage pipelined command may be issued from `C:\Program Files`, for example. It outputs the names of `.dll` files of size less than 10000 bytes in the current folder and all its subfolders. It wraps lines that are too long for the screen:

```
    Get-ChildItem -recurse -path *.dll | Where-Object {$_.length -lt 10000}
|
       Sort-Object -property Length | Format-Table -property name, length,
directory -wrap
```

The parameter `-path`, being default, can be omitted, as can the parameter `-property` in two of the stages. The command can be shortened further by introducing aliases and abbreviated parameters. For clarity, the statement is split by assigning an array object to the variable `$a`. (PowerShell variables have names that are text strings beginning with a dollar sign.) The cmdlet `Where-Object` (alias: ?) can recognize names of object properties without use of a

script block. That is, `{$_.length -lt 10000}` is equivalent to `length -lt 10000`. Thus.

```
    $a = ls -r *.dll |? length -lt 10000  # Store in $a names of all .dll files from current directory
                                          # downward with file sizes < 10000 bytes
```

The array object `$a` is piped to the alias `sort` of the cmdlet `Sort-Object` and the sorted object goes to `Format-Table`:

```
    $a | sort length | ft  name, length, directory -w # Sort entries of array $a on file size (length) and
                                                     # tabulate name, length, and directory
```

Finally, in one statement:

```
    ls -r *.dll |? length -lt 10000 |sort length |ft name, length, directory -w
```

where `ls` is the Unix-like alias of `Get-ChildItem`.

To contents

8More examples of pipelines

In PowerShell the "<" operator is reserved for future use, so that the cmd-mode/Unix redirection:

```
    .\program.exe < input.txt
```

does not work. Instead, pipe the input:

```
    cat input.txt | .\program.exe
```

The ">" operator redirects to output and ">>" appends to output, just like they do in cmd-mode and Unix.

List properties (names and values) of the file object `PSnotes.txt`:

```
    ls PSnotes.txt |fl *                          # Lists all properties: Directory, LastAccessTime, Basename, etc.
    ls PSnotes.txt |fl LastAccessTime, Basename   # Lists two properties: date/time of last access and file name.
```

If you want to tabulate (instead of list) names and values of one or more properties then pipe to `ft`. Example:

```
    ls PSnotes.txt |ft LastAccessTime, Basename   # Tabulates date and time of last access and file name
```

The difference between *tabulating* (ft) and *listing* (fl) properties is minor.

To list the content of the lines in `foo.out` that begin with at least four spaces together with their sequence numbers use `select-string` (alias `sls`). The cmdlet has the parameter `-pattern` that specifies a regular expression (regex):

```
    sls foo.out -pattern '^\s{4,}' | ft linenumber, line
```

(An empty line may not contain spaces and is then not shown by this command).

The string 'was' is found in all .txt files in the present directory (folder) by application of sls:

```
sls *.txt  -patt 'was' |ft -wrap filename, linenumber, line
```

Here -wrap indicates that line is not truncated at the width of the screen, but wrapped to the next line.

Find all directories (-dir) called Winx from the present directory downward (-rec). Inspect also hidden directories (-force) and suppress error messages (-ea 0):

```
ls Winx -dir -rec -force -ea 0 |ft
```

To contents

9The cmdlet Where-Object

The cmdlet Where-Object, which only appears in pipelines, has alias "?". The invocation of Where-Object must be followed by a *boolean expression* involving one or more members of the input object piped into the cmdlet. When the expression results in the boolean True the *full* input object is passed *unchanged* to the next stage of the pipeline. When it is False, neither the object nor any of its members are passed on. The cmdlet loops over all elements in the intermediate buffer created in the previous stage, and hence filters out the objects that do not pass the test.

In its simplest form Where-Object can be used to determine the existence of certain properties.
Example:

```
ls * |? -Property VersionInfo
```

The parameter (flag) -Property is default and may be omitted. When the length of VersionInfo is not zero, True is returned. In effect this statement lists all objects in the current directory that posses the property VersionInfo. It so happens that only files have this property and directories do not. So, only objects associated with files are passed along. Since this stage is the final stage, the file objects will printed and the directory objects will not. In summary, this statement is equivalent to ls * -file.

As another example, list the basenames ending with the letter r in the current directory. Use the Where-Object flag -Match (that has all the properties of the comparison operator -Match) with a regular expression (regex) and recall that the symbol $ indicates the end of a string in a regex:

```
ls  |? basename -Match 'r$'
```

This filters out the files and directories that do not have basenames ending on 'r'. The same object information (mode, last write time, length, name) is printed as by ls *. If only the basename is to be listed, pipe further to ft that selects this property:

```
ls  |? basename -Match 'r$' |ft basename
```

Parameters (flags) of Where-Object that can be used in the boolean expressions are: -Contains, -EQ, -GE, -GT -In, -Like, -Match, and -Is. All have a negated counterpart,

e.g. `-NotLike` and `-NE`, and almost all have a case-sensitive version, e.g. `-CNotLike`. An example of the parameter `-In` follows. The pipeline in the example starts with `ps`, which is an alias of `Get-Process`, and it limits the list of 'svchost' and 'firefox' processes to the first 20 in total. The paged memory sizes of these processes are tabulated together with the names of the processes.

```
    ps |? ProcessName -In  "svchost", "firefox" | select -f 20 |ft
processname,  PagedMemorySize
```

Invocation of `Where-Object` can also be followed by a script block: one or more statements (separated by semicolons) enclosed by curly brackets that return a boolean. Inside the block an object property named `PropName` is referred to by `$_.PropName`. Logical operators, such as `-and`, `-or`, and `-not` are not defined as parameters of `Where-Object` and can be used only in script blocks.

A final example uses a logical operator—and hence needs a script block—to find filenames (including extensions) that do not start with `c` and do not end with `t`. The script block contains `$_.`, a placeholder of the object name. The comparison operator `-NotMatch` uses a regex and `-NotLike` uses the wildcard `*`.

```
    ls |? {$_.name -NotMatch 't$' -and $_.name -NotLike 'c*'}     # Names
not ending on "t" or beginning  with "c"
```

To contents

10The cmdlet Select-Object

The cmdlet `Select-Object` (alias `select`) has two modes of operation:

1. Use with flags `-First, -Last, -Unique, -Skip, -Index`.
2. Use with one or more member names of input objects.

In the first mode, input objects are passed in their entirety into the pipeline. In the second mode, only the selected members (which are also objects, but smaller ones) are passed. One sees most clearly the difference of the modes when one pipes to `gm` (an alias of `Get-Member`) and counts. For instance,

```
    (ps firefox | gm).count                        # --> 92
    (ps firefox | select -index 2 | gm).count      # --> 92
    (ps firefox | select Workingset | gm).count    # --> 5
```

The first command:

```
    ps firefox
```

gives:

```
    Handles  NPM(K)    PM(K)      WS(K)     CPU(s)      Id  SI ProcessName
    -------  ------    -----      -----     ------      --  -- -----------
        303      28    54268      79676       1,42    1604  34 firefox
                    ...
        244      16    26640      29996       0,05    6356  34 firefox
                    ...
```

```
    244      16    26712      29916       0,08  18984   34 firefox
```

(As many lines as there are active Firefox processes). The second command:

```
ps firefox | select -index 2
```

gives the third output line of `ps firefox` (counting from 0):

```
Handles  NPM(K)    PM(K)      WS(K)     CPU(s)      Id  SI ProcessName
-------  ------    -----      -----     ------      --  -- -----------
    244      16    26640      29996       0,05    6356  34 firefox
```

The third command:

```
ps firefox | select Workingset
```

gives:

```
WorkingSet
----------
  81018880
     ...
  30617600
```

(an array with as many entries as there are active Firefox processes).

By the second command the same object (with 92 members) is passed as by the first command. By the third command the much smaller object `Workingset` (with 5 members) is passed into the pipeline. It is parenthetically of interest to observe that the third command returns working set sizes in bytes, whereas the first and second command give kB [under the header `WS(K)`].

The `select` is often useful after a sort, because by the flags `-first` and `-last` it can pick the first or last members of a sorted list. For instance, if one would like to know which five of the running process eat up the most memory:

```
ps | sort workingset -desc | select -first 5
ps | sort workingset | select -last 5
```

(`sort-object` sorts in ascending order by default.)
In its simplest form `Select-Object` is interchangeable with `Format-Table (ft)`. For example, the following two statements generate the exact same output:

```
ls *.txt | select basename
ls *.txt | ft basename
```

So, we have here an example of two cmdlets that are not completely orthogonal, but show considerable overlap.

To contents

11The cmdlet ForEach-Object

The iteration cmdlet `ForEach-Object` (alias `%`) has as loop body a script block: statements separated by semicolons between curly brackets. The cmdlet is commonly used in pipelines. However, it can also be used outside a pipeline. In the following example `$A` is an array (introduced in a later section) and `ForEach-Object` loops over its elements, writing them one by one to the host (the console in an interactive session):

```
$A = 1, 2, 3, 4
%{Write-Host $A} # ->  1 2 3 4
```

The use of `%`, although syntactically correct, offers no advantages, because `Write-Host $A` gives the very same output.

In pipelines, on the other hand, the cmdlet `ForEach-object` can be very powerful. Many cmdlets in a pipeline take an intermediate buffer as input and loop *implicitly* over its elements. Loops can be made *explicit* by the use of `ForEach-Object`. Its advantage is the greater flexibility: elements of the buffer may appear in complex expressions and more than one statement may be in the body of the loop.

In the next example a boolean expression is encountered. The simplest command using such an expression is of the form:

```
if (boolean expression){true branch}
```

The boolean expression (surrounded by round brackets) returns `True` or `False`. The body of the true branch is a script block as is the body of `ForEach-Object`. Now,

```
Get-Alias   |% {if ($_.name -match '^s') {Write-Host $_.name ' stands for: ' $_.definition}}
```

This displays about 24 lines:

```
sajb  stands for:  Start-Job
        ...
si  stands for:  Set-Item
        ...
sleep  stands for:  Start-Sleep
        ...
swmi  stands for:  Set-WMIInstance
```

`Get-Alias` sends ca 160 objects (aliases) through the pipeline. All objects have the property `name`, which is matched against a regular expression. The regex checks if the first letter is 's' or 'S'. If true, the object property `name`, the string `' stands for: '`, and the object property `definition` are written.

Incidentally, a list of aliases starting with 's' or 'S' can be obtained in a standard format from the shorter `Where-Object (?)` statement, which also loops over the whole buffer:

```
Get-Alias |? name -match '^s'   # regex
Get-alias |? name -like 's*'    # wildcard
```

Or from `Select-String (sls)`, which loops, too:

```
Get-Alias |sls '^s'             # regex
```

where the sls flag `-Pattern` is omitted as it is default.
Still shorter:

```
gal s*                          # wildcard
```

More than one statement may be in the body of the `ForEach-Object` loop. Example: list the names and count the number of lines of the `.txt` files in the present folder:

```
ls *.txt  |% {Write-Host $_.name ' '  -NoNewline;  (cat $_).count }
```

Here `-NoNewLine` is a parameter of `Write-Host`. The cmdlet `cat` returns an array object with elements the lines of the `.txt` file. Any array object has the property `count`.

The explicit-looping construct has advantages over implicit-looping methods when the elements that are looped over have special formatting or processing needs. Say, one wants to list all processes that have been running for more than 20 seconds, but one would like to see the results with less decimals than the standard (implicit) loop over the piped output of `sort` gives. A standard pipeline (`ps` is an alias for `Get-Process`):

```
ps |? cpu -gt 20 |sort cpu -desc| ft name, cpu
```

outputs the following:

```
Name                 CPU
----                 ---
firefox       275.390625
firefox          246.875
powershell     79.796875
notepad++         60.625
explorer       53.203125
firefox        45.078125
GoogleDriveFS  34.828125
thunderbird     22.15625
```

Say, *one* decimal figure is all that is needed and the time unit (sec) must be given. In a section below the formatting operator `-f` is discussed. The last stage of the next pipeline has an explicit loop over the sorted elements; its body contains `-f`:

```
ps  |? cpu -gt 20 | sort cpu -desc |%{"{0,15:s}: {1,10:f1} [sec]" -f $_.name, $_.cpu }
```

This gives:

```
          firefox:      275.4 [sec]
          firefox:      247.3 [sec]
       powershell:       80.0 [sec]
        notepad++:       62.0 [sec]
         explorer:       53.5 [sec]
          firefox:       45.1 [sec]
    GoogleDriveFS:       35.1 [sec]
      thunderbird:       22.2 [sec]
```

A loop body can hold computation (`1kb` is a literal constant of value 1024):

```
ls *.txt |% { "{0,30:s} {1,5:f2}" -f $_.name, ($_.length/1kb)}
```

This gives the names and sizes (in kB) of all `.txt` files in the current directory.

```
            ascii chars.txt  1.44
              codepages.txt  7.69
             conversions.txt  3.65
            cp1252 chars.txt  2.16
             cp850 chars.txt  2.12
            django_notes.txt 81.46
```

```
largest_cities_germany.txt  0.84
```

An example of `foreach` as a language construct:

```
$letterArray = "a", "b", "c", "d"
foreach($letter in $letterArray){
   Write-Host -ForeGroundColor green $letter
}
```

Yet another example as language construct:

```
$ff = ps firefox                  # Usually there is more than one
firefox process active
foreach($p in $ff) {$p.starttime}
```

This gives something like (in Dutch):

```
zaterdag 8 juni 2019 07:47:44
zaterdag 8 juni 2019 13:53:41
zaterdag 8 juni 2019 07:47:30
zaterdag 8 juni 2019 07:47:28
zaterdag 8 juni 2019 15:21:39
zaterdag 8 juni 2019 13:42:46
zaterdag 8 juni 2019 11:52:45
```

Incidentally, the very same output is obtained by:

```
(ps firefox).starttime
```

whereas

```
ps firefox | ft starttime
```

gives the same info in a day-month-year time format.

To contents

12Select-String

The cmdlet `Select-String` (alias `sls`) was encountered above. Because it is a useful tool—and not an easy one as it depends heavily on regular expressions—five more examples of its use are given. As all modern computer languages and text editors, PowerShell supports regular expressions.

Consider as a first example the string

```
$URL = "https://131.174.138.39/~pwormer/teachmat/PS_cheat_sheet.html"
```

The following two statements are equivalent when a single string is parsed:

```
$parsed  = $URL |sls -pattern '([0-9.]+)(.*)$'
$parsed  = sls -input $URL -pattern '([0-9.]+)(.*)$'
```

These statements capture the IP address and the path from `$URL` and assign them to members of the object `$parsed`. This object has several members, one of them the property `matches`. The one-stage pipeline:

```
$parsed.matches
```

lists on the screen the names and values of all properties of the object `$parsed.matches`. It gives:

```
Groups   : {0, 1, 2}
Success  : True
Name     : 0
Captures : {0}
Index    : 0
Length   : 60
Value    : 131.174.138.39/~pwormer/teachmat/PS_cheat_sheet.html
```

The array `$parsed.matches.groups` contains objects that in turn contain the captures, i.e., the subexpressions in the regular expression that are between parentheses. The zeroth element gives the total string matched (as does the property `$parsed.matches.value`). In summary.

```
$parsed.matches.groups[0].value :
131.174.138.39/~pwormer/teachmat/PS_cheat_sheet.html
$parsed.matches.groups[1].value : 131.174.138.39
$parsed.matches.groups[2].value :
/~pwormer/teachmat/PS_cheat_sheet.html
```

By default, `Select-String` finds the first occurrence in a string only. If more occurrences must be found, use the flag `-AllMatches`.

As a second example find all 4 italic substrings in:

```
$str = "<i>Snakes</i> and <i>lizards</i> are icky, while  <i>cats</i>
and <i>firefoxes</i> are cute"
```

Use:

```
$regex = "<i>(.*?)</i>"
$M = $str | sls -AllMatches  $regex
```

The regex matches a string that is enclosed within `<i>` and `</i>`. It matches zero or more (`*`) arbitrary characters (`.`) and it captures the string matching the regex inside the parentheses. The question mark in `$regex` indicates a "lazy" match, the matching stops as soon as the next `</i>` is found. The sls flag `-Pattern` is omitted because from the context its is clear that `$regex` contains the pattern. The object `$M` has the property `Matches` with 4 entries corresponding to 4 matches. An entry in this array is the array `Groups` that contains the matches: `Groups[0].Value` contains the total match and `Groups[1].Value` is the capture.

```
$M.Matches[0].Groups[1].Value  # Snakes
$M.Matches[1].Groups[1].Value  # lizards
$M.Matches[2].Groups[1].Value  # cats
$M.Matches[3].Groups[1].Value  # firefoxes
```

or

```
$M |%{$_.Matches} |ft -HideTableHeaders {$_.Groups[1].Value}
-->
Snakes
lizards
```

```
cats
firefoxes
```

So far, a *single* string was entered as input to `Select-String (sls)`. Also files consisting of more than one line (such as source codes, or `.log` and `.txt` files) can be entered into this cmdlet. An (implicit) loop over the lines in the file is taken and the matching happens line for line. A collection of, e.g., `html` files can be piped into `sls`, as in the third example:

```
ls *.html |sls  "!DOCTYPE" |ft Path, Line
```

where the sls flag `-Patt` is again omitted.

As a fourth example we notice that the present file, called `PS_cheat_sheet.html`, contains strings that are enclosed between and , the strings to be displayed in bold face. A pipeline will be constructed that lists these strings. (The pipeline will *not* find bold text that extends over different lines). Define a regular expression (regex) that captures bold HTML text:

```
$reg = '<b>(.*?)</b>'
```

The regex `$reg` is used in:

```
sls -Path PS_cheat_sheet.html -Allm -Patt $reg |% {$_.matches}| ft -H {$_.groups[1].value}
```

which writes all the bold strings in the present notes. The flag `-Path` may be omitted, as it is default.

As a fifth and final example, consider a file named `foo.txt` with a somewhat messy layout. It contains positive, negative, and unsigned integers in columns 20...25 (six columns). The content of `foo.txt` is:

```
   101   xxxxx     -23    ddd
     20  ddd          +2     eee
 30    %^&&fghu     -100    ffff
   40    qawer       1000  fffqq
And yet one more      1    unsigned number in column 23!
```

The following line lists the sum (=880) of the numbers in columns 20-25:

```
 $s=0; (sls foo.txt -patt '.{19}([0-9-+]{1,6})').matches |% {$s+= $_.groups[1].value}; $s
```

The regex skips the characters in the first 19 columns and captures 1 to 6 digits and plus/minus signs in columns 20-25. The sum is accumulated in `$s` and note that PowerShell converts captured strings to integers in order to enable addition. Note that `foo.txt` is not piped into `sls`, but is an input file. This is possible as the sls flag `-Path` is default.

To contents

13Create an item

The cmdlet `New-Item` (alias `ni`) creates a new item. An item is one of the following:

- Folder (aka directory)
- File
- Link (symbolic or hard)
- Registry key
- Registry entry

As an example, create a symbolic link on the desktop, named `manual`. *Both PowerShell and CMD require admin privilege to do this.* The newly created link `manual` targets the file `manual.txt` that exists in the current folder:

```
    New-Item -ItemType SymbolicLink -Path C:\Users\User\Desktop\manual -
Target .\manual.txt
```

This can be shortened to:

```
    ni -ty sym  C:\Users\User\Desktop\manual -v manual.txt
```

The parameter `Target` has the alias `Value`. `ItemType` has the alias `Type`. Further `-Path` and the current folder indicator `.\` may be omitted.

Create more than one `.txt` file at once, example:

```
    'file1', 'file2',  'file3' |% {ni $_'.txt'}
```

Also more than one folder can be created in this way:

```
    'dir1', 'dir2',  'dir3' |% {ni -ty dir $_}
```

And remove by

```
    'dir1', 'dir2',  'dir3' |% {ri  $_}
```

`Remove-Item`, alias `ri`, does not have (or need) a flag `-ty`.

To contents

14Rename an item

The cmdlet `Rename-Item` (aliases: `ren` and `rni`) renames items such as files, folders, and registry keys. In contrast to the CMD command `rename`, it does not allow a wildcard in the name of the files. Although

```
    ren report.txt report.doc
```

is correct in CMD-mode as well as in PowerShell, the command that includes the wildcard `*`

```
    ren *.txt *.doc     # Error in PS!
```

only works in CMD-mode. PowerShell returns an error.

The cmdlet `Rename-Item (ren)` can take piped input for the old name and recognizes the flag `-newname` for the new name. For example,

```
    'report.txt' | ren -newname report.doc
```

gives the required change of the file extension.

To change multiple names at once, one may use the operator `-replace`. Its syntax is:

```
    string -replace regex, new_name
```

In `string` every substring matched by the regular expression `regex` is replaced by `new_name`. For example.

```
'123xyz456' -replace '[a-z]{3}', 'abc'   # --> 123abc456
```

To replace all file extensions `.pdf` by `.qdf` in the current directory, the regular expression: `'\.pdf$'` may be used. In this regex `pdf` is matched at the end of the string by the anchor "`$`". The dot is escaped so that its meaning is not the regex arbitrary character, but the ordinary punctuation mark. Thus,

```
ls *.pdf | ren -new { $_.name -replace '\.pdf$', '.qdf' }
```

The value of the script block (the part between curly brackets) is a string: the new file name.

The PowerShell statement in the previous example is considerably more difficult to memorize than the equivalent CMD statement. Therefore, it is useful to mention that `cmd /c` invokes a CMD command from within PowerShell (the flag `/c` stands for "command"). Thus.

```
cmd /c rename *.pdf *.qdf
```

changes the extension `.pdf` to `.qdf` for all items in the current folder. Parenthetically, the following PS command achieves the same task without invoking `-replace`:

```
(ls *.pdf).basename |% {ren .\$_.pdf .\$_.qdf}
```

For some reason it is necessary in this specific statement to indicate the present directory by the prefixes `.\`, while in:

```
ren report.pdf report.qdf
```

the prefix is not needed. However, this statement is not so easy to memorize either.

A caveat for readers not well acquainted with regular expressions: context is important. For instance, the string `'.txt'` has a different meaning as a regex than as an ordinary string. As a regular expression it matches a substring consisting of an arbitrary character concatenated to the string `txt`. Noting that `-match` takes a regex as second operand, while `-eq` simply compares strings (see Learn MicroSoft for a list of the comparison operators and their details), one can understand the following:

```
'Qtxt' -match '.txt'  # --> True
'Qtxt' -eq    '.txt'  # --> False
```

To contents

15Typecasting, strings, constants, and arrays

Type cast operators are among others: `[int]`, `[long]`, `[string]`, `[char]`, `[bool]`, `[byte]`, `[double]`, `[decimal]`, `[single]`, `[array]`, `[xml]`, `[hashtable]`, and `[PSCustomObject]`. Many have an array variant, e.g., `[int[]]$irray` casts `$irray` as an integer array.

Once the type of a variable is cast explicitly, an *implicit* recast is forbidden. Examples:

```
    [string]$s = 'Peanuts'    # Explicit casting of $s to type string
    $s         = 4            # Assign to $s new string '4'
    $s * 3                    # Gives 444 (triplicates the string)
    [int]$s    = 4            # Explicit recasting of type (to int32)
    $s * 3                    # Gives 12
    $s         = 'bear'       # Error: Cannot convert value "bear" to type
"System.Int32"
    [string]$s = 'bear'       # Explicit recasting is OK
    $s = [char]0x07           # Recasting on RHS is OK
```

Note that `[char]0x07` (ASCII 7) sounds the bell.

Another type casting example:

```
    'a', 'b', 'c', 'd' > letters.txt       # Each letter on its own line
into file letters.txt. Output in UTF-16 (!)
    [string]$letter = cat letters.txt      # Get-Content (alias cat),
assign to string $letter
    $letter                                # -> a b c d   (a single string)
    rv letter                              # Remove-Variable (alias rv)
$letter (no $ prefix in rv!)
    $letter = [string](cat letters.txt)    # -> a b c d  (brackets are
necessary to cast the output of cat)
```

Strings are as in the computer language PHP: 'Singly' quoted strings: no expansion of variables or escape character (backtick). "Doubly" quoted: expansion of variables. Expressions under `$` are evaluated. For example.

```
       $five  = 5
    '3*$five'  -> 3*$five
    "3*$five"  -> 3*5
    "$(3*5)"   -> 15
```

A backtick escapes between double quotes. For example, escaping the dollar symbol: `"`$(3*5)"` gives `$(3*5)`. Backticks are unchanged under single quotes: `'`$(3*5)'` gives `'`$(3*5)'`.

Notes:

1. `"`n"` gives the newline character.
2. The string `'John Doe'` can be appended to a file simply by `'John Doe' >> out.txt`. A single angular bracket (>) overwrites. The out file is in Little-Endian UTF-16 encoding:

```
3.     'John Doe' > try1.txt   # 22 bytes  (UTF-16LE: 2 byte/char + 2
   bytes BOM + 4 bytes EOL)
```

4. The cmdlet `add-content` (alias `ac`) writes to file by default in the system's active code page, and allows specification of other encodings. The system's current code page is shown by `chcp`.

```
5.     ac try.txt 'John Doe'   # 10 bytes  (ASCII:  1 byte/char + 2
   bytes EOL)
```

Strings have methods. Enter

```
    "This is a valuable string" |gm
```

and lots of string methods are shown, among which is `split`. That is.

```
    $a = ("This is a valuable string").split(' ')
```

returns the array `$a` with 5 members, the strings `"This", "is", "a", "valuable", "string"`. The method may be used in:

```
    $env:path.split(';')
```

which returns the Windows environmental variable `path` with its entries stacked vertically.

Here-strings

`@' ... '@` (no expansion) and `@" ... "@` (with expansion). The openings `@'` and `@"` must start in column 1 and be on a single line. The same holds for the endings `'@` and `"@`.

Example, assume `$a -eq "Big Brother"`:

```
@'
    $a
    $(3*5)
'@
```

gives

```
    $a
    $(3*5)
```

while

```
@"
    $a
    $(3*5)
"@
```

gives

```
    Big Brother
    15
```

Constants

Use cmdlet `Set-Variable` (alias `set`) to assign constants:

```
    set pi 3.14 -opt const    # Assign constant (no $ in set pi !)
    $pi                       # Gives 3.14  (Use $ in reference to the
constant, just set without $!)
    $pi = 2                   # -> Error: Cannot overwrite variable pi
because it is read-only or constant.
    set pie $([math]::pi) -opt const  # Note $(..) around
    $pie                      # -> 3.14159265358979
```

Arrays

Refer to element `i` of array `A` by `A[i], i=0,1,..`

```
    $A = @('a', 'b')          # Array  (round brackets, commas or
semicolons as separators)
```

```
    $A += 'c'                   # Push 'c'
    $A += 'd'                   # Push 'd'
    $A                          #  -> a\n b\n c\n d  (vertical stack)
    $A.GetType()                #  -> True        Object[]      System.Array
```

Round brackets and `@` are not necessary:

```
    $arry = 1, 2, 3, 4, 5      # Array, $arry[0] is 1, $arry[$arry.length-1]
is 5.
    $arry.GetType()            # -> True         Object[]      System.Array
```

An array with the consecutive numbers 5, .., 15 is generated as follows:

```
    $seq = 5..15
    $seq                       # -> 5, .., 15 (stacked vertically)
```

To contents

16The formatting of strings

The easiest way of formatting strings is by expanding doubly quoted strings. Example:

```
    $brides_name = "Sue"
    $grooms_name = "Roger"
    $announcement = "The beautiful bride, $brides_name, and the cool groom,
" +
        "$grooms_name, were wedded today"
    $announcement
```

which outputs:

```
    The beautiful bride, Sue, and the cool groom, Peter, were wedded today
```

The simple expansion technique is not formatting *per se*. Instead, insertion of spaces, rounding and aligning numbers, etc., can be done by application of the format operator `-f`. Schematically the binary operator `-f` is used as follows:

```
    "String containing format-items"  -f  array with elements to be
substituted on lhs
```

On the left-hand side of the operator `-f` is a string containing format-items, which are of the form:

```
    {index[,alignment][:formatString[number]]}
```

(Here square brackets surround optional values).

- The `-f` operator allows on its left-hand side both singly and doubly quoted strings.
- The index is 0-based and refers to the position in the array on the right-hand side of `-f`.
- The alignment gives the total number of spaces reserved for the output, with the output string right-aligned in the reserved space. If the alignment is too small, the system takes the spaces it needs.
- After the colon is the format string (see below). In the case of floating point numbers, the number of decimals is given by the number after the format string. By default it is 2. The number of digits before the decimal point is adapted by the system so as not to lose information.

- The currency and decimal symbol depend on the Windows country/region settings.

Examples:

```
    '{0} rounded to  4 decimals is {0,0:f4}' -f 12.3456931    # ->
12.3456931 rounded to  4 decimals is 12.3457
    "12345 converted to hex is: {0,10:x}"    -f 12345         # -> 12345
converted to hex is:       3039
    'Prices are {0,0:c} and {1,8:c0}'        -f 12.3998, 1.99 # -> Prices
are € 12.40 and      € 2
```

`Write-Host` can write formatted strings by the use of the `-f` operator (the host is usually the console):

```
    $str = "One decimal:{1,5:n1}; two decimals:{0,7:n2}; three
decimals:{2,10:n3}"
    Write-Host($str -f 2.141, 1.141, 3.141)
```

Note the order: second array element first, then the first, and finally the third. Also notice the spaces dictated by the alignment parameter:

```
    One decimal:  1.1; two decimals:   2.14; three decimals:     3.141
                |||||               |||||||                 ||||||||||
```

The following is an (incomplete) list of format strings; optionally some may be appended by an integer giving the number of decimals:

:c
: Currency (output depends on the country/region set in Windows Control Panel: `$, €, ¥, ...`).

:d
: Decimal. Only for integers (no decimals).

:e
: Scientific (exp) notation.

:f
: Fixed point, `n:fd` gives field of length `n` and `d` digits after the decimal point. If `n` is too small the system adapts.

:g
: Most compact generic format, fixed or scientific.

:n
: Number, includes decimal and thousands separators (choice depends on the country/region set in Windows Control Panel).

:p
: Percentage.

:x
: Hexadecimal format (only integers).

Formatting may be applied in pipelines. For instance, to list basenames and their lengths (kB) in a pretty format (round brackets in the array on the right-hand side of `-f` are needed to give priority to the division), use:

```
    ls *.pdf |%{"{0,31} {1,6:f1} kB" -f $_.basename, ($_.length/1kb)}
```

This gives:

```
            angmom  254.9 kB
```

```
      correlation  318.2 kB
           Renner  151.7 kB
    Renner_german 1070.6 kB
         rigrotor  272.9 kB
        rungelenz  188.0 kB
```

To contents

17Hash tables

Hash tables are also known as "dictionaries" or "associative arrays".

```
    $assoc = @{one=1; two=2}  # Hash table (curly brackets, semicolons as
separators)
    $assoc.one                # -> 1
    $assoc['two']             # -> 2
    $assoc['three'] = 3       # Adds a member, note the square bracket
notation
    $assoc.four = 4           # Adds a member, note the dot notation
    $assoc.GetType()          # --> True         Hashtable
System.Object
    $assoc                    #
```

The last statement (`$assoc`) gives:

```
    Name                           Value
    ----                           -----
    four                           4
    one                            1
    three                          3
    two                            2
```

It is a coincidence that the keys in this list are alphabetically ordered, the items in a hashtable are intrinsically unordered. Keys of a hashtable (= associative array) must be unique, a certain key cannot occur more than once.

When hashtables are tabulated, for instance by the cmdlet `Format-Table` (`ft`), their format is not very illuminating, as their headings are simply `Name` and `Value`. For instance.

```
    $Canada = @{Country = "Canada"; Population = "40010000";
Capital="Ottawa"}
    $Canada | ft
```

prints

```
    Name                           Value
    ----                           -----
    Population                     40010000
    Capital                        Ottawa
    Country                        Canada
```

This output can be prettified by converting the hashtable to a PSCustomObject. This can be done by the recast (a loop over one element):

```
$C = $Canada | % {[PSCustomObject] $_}
```

This statement converts the keys and values of the hashtable `$Canada` to properties and values of the object `$C`. They can be printed:

```
$C | ft
```

This gives, (note the headings):

```
Population Capital Country
---------- ------- -------
40010000   Ottawa  Canada
```

Nice tables, with meaningful headers, can be created by recasting existing arrays to PSCustomObjects. As an example, consider three mutually matching arrays, each with four entries:

```
$Countries   = @('Canada', 'US', 'Germany', 'France')
$Populations = @('40 million', '333 million', '84 million', '68 million')
$Capitals    = @('Ottawa', 'Washington DC', 'Berlin', 'Paris' )
```

To obtain a single table containing this statistical info, an auxiliary array `$Stats` is created with 4 elements:

```
$Stats = 0..3|%{@{Country = $Countries[$_]; Population=$Populations[$_]; Capital = $Capitals[$_] }}
```

The elements of `$Stats` are hashtables. For instance, the last assignment in this loop creates the hashtable with three entries:

```
$Stats[3] = @{Country='France'; Capital='Paris'; Population='68 million'}
```

The array `$Stats` is converted to an object containing a table with all info:

```
$Stats_obj = $Stats | %{[PSCustomObject] $_}
```

Because a single array takes part in the recast to a PSCustomObject, this is interpreted as a single statement. In the first step, through the `ForEach-Object` loop, the object `$Stats_obj` is created with the properties `Country`, `Population`, and `Capital` and the respective values of `$Stats[0]` are assigned to these properties. In the next steps the values of `$Stats[i], i=1..3` are added to the newly created properties of `$Stats_obj`. These properties are arrays that contain the country names, capitals, and populations of the four countries.

It possible to cut out the auxiliary array `$Stats` altogether by replacing `$_` in the previous statement by the definition of `$Stats`, thus creating the PSCustomObject `$Stats_obj` by the following statement:

```
$Stats_obj = 0..3| %{[PSCustomObject] @{Country=$Countries[$_]; Population=$Populations[$_]; Capital=$Capitals[$_] }}
```

In any case, the command:

```
  $Stats_obj | ft Country, Population, Capital
```

gives finally the nice table:

```
Country Population  Capital
------- ----------  -------
Canada  40 million  Ottawa
```

```
US      333 million Washington DC
Germany 84 million  Berlin
France  68 million  Paris
```

To contents

18A practical example: a table of WiFi passwords

After a Windows laptop is connected to a WiFi network, Windows remembers the network name (the SSID, short for "Service Set IDentifier"). When the network is protected (for instance by WPA-2) and a password is needed, Windows stores the requested password, too. The advantage of storing this info is obvious: when the same laptop is reconnected to the same network at a later time, this reconnection is (or can be set to be) automatic.

From a PowerShell session in *administrator mode*, the SSIDs and their passwords, stored on the computer on which the session is running, can be obtained. In order to extract this info, one goes from the PowerShell environment to the subenvironment `netsh` by simply entering `netsh`. From there the subsubenvironment `wlan` is entered by:

```
netsh>wlan
```

In the wlan environment one can obtain all the stored SSIDs by issuing the command:

```
netsh wlan>show profiles
```

Its output contains several lines among which the following contain the SSIDs:

```
All User Profile     : SitecomAFA7DC
All User Profile     : ZiggoFD1ECD1
All User Profile     : BHNTG1682G9BF2
All User Profile     : Hotspot_Maasduinen
All User Profile     : Softstar-WiFi
```

When the plain password of one these SSIDs is required, for example of `ZiggoFD1ECD1`, one enters:

```
netsh wlan>show profile ZiggoFD1ECD1 key=clear
```

Here `key=clear` requests the plain, unhashed, password. Many lines are put out to the screen, among which the line

```
Key Content            : 531@1RMQA%9
```

where the string after the colon is the password of `ZiggoFD1ECD1` (this password is fictional of course). The steps can be automatized and a table of SSIDs with passwords can be generated automatically.
First it is noted that commands of a subenvironment can be issued from PowerShell directly. Thus,

```
PS> netsh wlan show profile ZiggoFD1ECD1 key=clear |sls 'Key Content'
```

gives the password belonging to the SSID `ZiggoFD1ECD1`:

```
Key Content            : 531@1RMQA%9
```

To obtain a list of all stored passwords, we must loop over all SSIDs stored on the laptop. Suppose we have collected these SSIDs in an array $SSIDs, then

```
$SSIDs | %{netsh wlan show profiles $_ key=clear} | sls 'Key Content'
```

gives a set of `Key Content` lines that contain the passwords. If we want to capture the passwords we modify the sls pattern to:

```
'Key Content \s*:\s(.+)$'
```

and loop over the captures

```
$SSIDs | %{netsh wlan show profiles $_ key=clear} | sls 'Key Content
\s*:\s(.+)$' |
            %{$_.matches.groups[1].value}
```

This gives the required (fictitious) list of passwords:

```
z312_2215A
531@1RMQA%9
Sugar_Mill
1
MaldenDept
```

(The password 1 is of the "Hotspot_Maasduinen", which is unprotected).

To obtain the array `$SSIDs`, we proceed in a very similar way with the addition that the captured SSIDs are put into the array `$SSIDs`:

```
$SSIDs = @( netsh wlan show profiles |sls ':\s(.+)$' |
         %{$_.matches.groups[1].value})
```

To obtain, finally, a nice table, the passwords of the individual networks are collected into the array `$PSWDs` by:

```
$PSWDs = @($SSIDs | %{netsh wlan show profile $_ key=clear} |
         sls 'Key Content \s*:\s(.+)$'| %{$_.matches.groups[1].value})
```

The arrays `$SSIDs` and `$PSWDs`are used to compose hashtables with keys `SSID` and `Password`. The hashtables are added one by one to the properties `SSID` and `Password` of a PSCustomObject and the total is piped to `ft`:

```
$n = $SSIDs.length - 1
0..$n| %{[PSCustomObject] @{SSID=$SSIDs[$_]; Password=$PSWDs[$_]}}
```

This yields the table:

```
SSID                Password
----                --------
SitecomAFA7DC       z312_2215A
ZiggoFD1ECD1        531@1RMQA%9
BHNTG1682G9BF2      Sugar_Mill
Hotspot_Maasduinen  1
Softstar-WiFi       MaldenDept
```

All this can be collected into the one horrifically looking statement:

```
netsh wlan show profiles | sls ':\s(.+)$' |
```

```
%{$SSID=$_.Matches.Groups[1].Value; $SSID} |
%{(netsh wlan show profile $SSID key=clear)} |
sls 'Key Content \s* :\s(.+)$' |
%{$_.Matches.Groups[1].Value} |
%{[PSCustomObject]@{ SSID=$SSID;Password=$_ }} |ft
```

that yields the very same table of SSIDs and their associated passwords as shown above.

To contents

19Comparison

Comparison operators are among others: `-eq, -ne, -gt, -ge, -lt, -le, -like, -notlike, -match, -notmatch` and `-cmatch`. Although the operator `-replace` does not perform a comparison, it is usually included in this group.
Examples:

```
'peanutbutter' -like 'nut'          # false
'peanutbutter' -like '*nut*'        # true   (* is wildcard)
'peanutbutter' -notlike '*nut*'     # false
'peanutbutter' -notlike 'nut'       # true
'peanutbutter' -match '[a-z]+'      # true   (regex: all letters)
'peanutbutter' -match 'r$'          # true   (regex: last letter is r)
'peanutbutter' -match '[A-Z]+'      # true   (PS is case insensitive)
'peanutbutter' -cmatch '[A-Z]+'     # false  (cmatch  matches cases)
'peanutbutter' -replace 'u', 'U'    # 'peanUtbUtter'
```

The operators `-match, -cmatch and -notmatch` compare against a regular expression, similar to:

```
$M = Select-String -input string  -Pattern sls_regex
```

As shown earlier, subexpressions between round brackets in `sls_regex`, are captured in `$M.matches.groups[k].value` with `k=0,1,..`

The comparison operators handle captures, optionally defined in their regexes, differently than `Select-String`. Each time they have a single variable as input, and the `-match` result is `True`, (or `-notmatch` results in `False`), they overwrite the `$Matches` automatic variable. `$Matches` is a hashtable that always has a key named '0', which stores the entire match. If the regular expression contains captures, `$Matches` contains an additional key for each capture. Note that this hashtable contains only the first occurrence of any matching pattern, there is nothing similar to the `-Allmatches` flag of `sls`.
Example:

```
$URL = "https://131.174.138.39/~pwormer/teachmat/PS_cheat_sheet.html"
$URL -match '([0-9.]+)(.*)$'    # -> True
```

The command `$Matches` gives:

```
Name                           Value
----                           -----
2                              /~pwormer/teachmat/PS_cheat_sheet.html
```

```
1                131.174.138.39
0                131.174.138.39/~pwormer/teachmat/PS_cheat_sheet.html
```

For an application of a comparison operator in practice, notice that websites that offer downloads sometimes publish the SHA256 hashes of their downloads. Suppose a download is called `DownLoad.msi` and its hash is called $hash (a string containing 64 hex characters from [A-F0-9]). It is assumed that the variable `$hash` can be obtained from the website. Then the cmdlet `Get-FileHash` can be used in conjunction with the comparison operator `-eq`:

```
(Get-FileHash .\DownLoad.msi).Hash  -eq $hash
```

If the result is `True`, the file `DownLoad.msi` can be trusted.

To contents

20Builtin classes

PowerShell has access to .NET classes, one is `[console]` with methods (among others) `beep` and `write`: see the MSDN (microsoft developer network) site.

```
[console]::beep(800, 1000)     # beep at 800 Hz for 1000 msec
[console]::write([char]0x07)   # Write hex 07, that is, ring the bell
(does not work under ISE)
[console]::readkey()           # Return name of key + modifier(not under
ISE)
```

Another built-in class is `[math]`. See MSDN.
Examples:

```
[math]::pi                     # 3.14159265358979
[math]::cos([math]::pi)        # -1
[math]::max(-1,  -4)           # -1
[math]::pow(10, 3)             # 1000
```

To see the contents of `[math]`:

```
[math] | gm -static
```

To contents

21Random numbers

PowerShell knows about (pseudo) random numbers:

```
Get-Random -min 0.0 -max 1.0  # Random number between 0.0 and 1.0,
```

see the help of `Get-Random` for the bounds.

Alternatively, by the use of `New-Object` one can create the PS object `$Rand` as an instance of `Random` (a System.Random object):

```
$Rand = New-Object Random
$Rand | gm                # Gives methods of $rand, among which
NextBytes
```

The creation of a byte array by type casting is equivalent to the creation of a System.Byte object (a byte array of length 4 plus methods):

```
    [byte[]]$out = @(0,0,0,0) <-->  $out = New-Object Byte[] 4
```

Overwrite zeroes in the array with random bytes by a method of instance `$Rand`:

```
    $Rand.NextBytes($out)     # Fill array $out with 4 integer random
numbers n: 0 ≤ n ≤ 255
    $out                      # To console
```

To contents

22Errors

Almost all cmdlets recognize the flag `-ErrorAction`, abbreviated as `-ea`. The parameters of this flag (`Continue`, etc.) may be replaced by numbers, as follows:

```
    # -ErrorAction Continue | Ignore | Inquire | SilentlyContinue | Stop
    # -ea            2      |    4   |    3    |       0          |   1
```

For instance, suppress error message about inaccessible subdirectories as follows:

```
    ls -rec -ea 0 *.dll # all .dll files in present and all subdirectories
```

Errors may be trapped. Enter `Get-Help about_trap` to see how. The very same info in the form of a webpage is here: About trap.
Example of trapping:

```
    Trap [System.Exception] {
       "Command error trapped.`n  " # Automatic variable '$_' contains
system error msg; `n gives newline,
       continue                    # Suppress traceback, continue after
erroneous statement
    }
    nonsenseString                 # Erroneous statement: unknown cmdlet,
function or script.
    'Execution continues'
```

PS also has a `Try ... Catch` construct:

```
    try {
        An error                   # Illegal statement
    }
    catch {
       "An error occurred"
    }
```

The global object `$error` is a stack containing the consecutive non-trapped errors. To list the latest and the first error, respectively:

```
    $error[0] | fl                  # The latest
    $error[$error.count-1] | fl     # The first
```

To contents

23Functions

A *PS function* is written in the PowerShell script language and is not compiled, but interpreted. Often functions are written by end-users. In contrast, a cmdlet is written in a .NET programming language, such as `C#` ("C sharp"), and is an intrinsic part of PS. A function name, just like a cmdlet name, is preferably of the form "verb-noun", where "verb" should be from a set of recommended verb names. To get this set, use `get-verb`. For recommended verbs starting with `w`, use `get-verb w*`.
To list the verbs used in all aliases, functions (including user functions), and cmdlets sorted in alphabetical order, issue:

```
Get-Command |Select-Object verb | Sort-Object * -unique
```

The functions of PS are invoked with parameters trailing the invocation, with no commas between them, no round brackets behind the function name, and where flags are optional, schematically:

```
  Function_name [-flag1] param1 [-flag2] param2 ...
```

Functions can be defined with "classical" arguments.
Example:

```
function Write-ToScreen($path, $filename) {
    Write-Host("path = {0:s} filename = {1:s} " -f $path, $filename)
}
```

which is called as:

```
Write-ToScreen "C:\User\documents"  "manual.txt"     # No commas, no brackets!
```

Clearly, without flags the order of the parameters matter. Slightly less "classical" is the following possible definition, where flags are defined with default values:

```
function Write-ToScreen($path='c:\', $filename="*.txt") {
    Write-Host("path = {0:s} filename = {1:s} " -f $path, $filename)
}
```

Example of invocation:

```
Write-ToScreen  -filename manual.pdf   # --> path = c:\ filename = manual.pdf
```

Alternatively, function arguments can be defined with flags by the `param` statement.
Example:

```
function Write-ToScreen{
    param($path, $filename)
    Write-Host("path = {0:s} filename = {1:s} " -f $path, $filename)
}
```

This function is invoked (the flags can be abbreviated to unique strings) as:

```
Write-ToScreen  -filename "manual.txt"   -path "C:\User\documents"
Write-ToScreen  -f "manual.txt"   -p "C:\User\documents"
```

Or more briefly, where the parameters depend on position:

```
    Write-ToScreen "C:\User\documents"  "manual.txt"
```

The `param` form of a function cannot be mixed with the "classical" form: `Function_name (arg1, arg2, ..)`

With regard to return values: (most) console output generated in the body of the function is collected into an array which is returned. The following three kinds of console output are collected in the return array:

- `Write-Output`.
- A simple string reference.
- The end of a pipeline (including a pipeline of one segment).

After completion of the function call, the return array may be assigned to a variable, written to the console (the default), redirected to a file, or entered into a pipeline. Example:

```
    function list-no{
        write-output "first"
        "second"
        "third" |fl
   }
    $a = list-no     # Nothing to the console; output collected in return
array $a
    $a               # --> first\nsecond\nthird  (vertically stacked)
```

Not *all* console output is collected: the cmdlets `Write-Host` and `Out-Host -i` write *only* to the console during the execution of the function; they do not generate return values:

```
    function list-yes{
        Write-Host "first"
        Out-Host -input "second"
    }
    $a = list-yes    # --> first\nsecond  (vertically stacked) to console
    $a.length        # --> 0
```

Because the return array may be redirected to a file and not all console output generated during function execution ends up in this array, the return values require close inspection. Example:

```
    function Write-Vars{
        $a, $b, $c, $d, $e = 'A', 'B', 'C','D', 'E';
        Write-Output $a           # To return array (including EOL chars)
        $e | fl                   # To return array
        Write-Host $b             # To console
        $c                        # To return array
        Out-Host -i $d            # To console
    }
    $f = Write-Vars      # 'B', 'D' to console; 'A', 'E', 'C' to $f
    $f                   # 'A', 'E', 'C' to console
    Write-Vars           # 'A', 'E', 'B', 'C', 'D' to console  (in order of
assignment).
    Write-Vars > foo.out # 'B', 'D' to console; 'A', 'E', 'C' to foo.out
```

One could expect that the statement `Write-Vars` would write first the immediate values of `$b` and `$d` followed by the values of the return array, but this is not the case.

More examples:

```
function list-txt{
    ls *.txt      # Output object into pipeline
}
$a = list-txt  # No console output, object in pipeline to $a.
$a             # Info of .txt files to console (same as ls *.txt )
$a  | ft name  # Only file names to console

function list-txt{
    ls *.txt|write-host
}
$a = list-txt  # File names (including paths) of .txt files to console, nothing to $a.
$a             # No output, no returned parameters.
```

Function parameters can have a default value:

```
function f{
    param($c = "Pete")
    $c
}
f          # -> Pete
f Sue      # -> Sue
f -c John  # -> John
```

Arguments are passed in the automatic array `$args`:

```
function Get-Arg{
    foreach ($p in $args) {
        Write-Host $p
    }
}
Get-Arg  1, 2, 'pink elephant',  icecream      # --> 1 2 pink elephant icecream
```

Functions may handle piped input object (property names of object must be known in advance—as always in pipelines). The named members of an object or hashtable piped into the function are handled in a `Process` block within curly brackets.
Example:

```
Function Test-PipedValue{
    Process {
        Write-Host ("name: {0}; color: {1}" -f  $_.name,  $_.color)
    }
}
```

Create a hashtable and pipe into function `Test-PipedValue`:

```
$person = @{name = 'Jean'; color = 'Black'}
$person | Test-PipedValue   # --> name: Jean; color: Black
```

The `Process` command loops over the elements (inside the `Process` block referred to by `$_`) of the piped object, as shown in the next example:

```
function Test-Loop{
    Process {
        Write-Output $_    # Appends EOL char to each element
    }
}
```

Example of invocation:

```
1, 2, 3, 4 | Test-Loop      # --> 1\n 2\n 3\n \4 (vertical stack)
```

See `help about_functions` for more info.

To contents

24Switch

The following `switch` statement is case sensitive because of the flag `-casesensitive`:

```
function f ($str) {
    Switch -casesensitive ($str) {
        'aap'  { write-host 'AAP'  }
        'noot' { write-host 'NOOT' }
        'mies' { write-host 'MIES' }
        'wim'  { write-host 'WIM'  }
        Default { "Unable to determine value of $str" }
    }
    "Statement after switch"
}
f noot    # --> NOOT \n Statement after switch
f Noot    # Unable to determine value of Noot \n Statement after switch
```

To contents

25Scripts

A script is a collection of PowerShell commands contained in a file with extension `.ps1`. The script is invoked from a PS session by entering its file name (= script name) prefixed by its path. If the script has parameters (defined in a `param` statement) their values follow the script name, optionally prefixed by the parameter names, just as is the case in a function invocation.

The output of scripts is comparable to that of functions. That is, the command `Write-Host` writes immediately (and only) to the console, while `Write-Output` writes to a return

array. After termination of the script it is decided what happens to the return array: redirected to a file or to the console. Create the script `Measure-Text.ps1` containing the following 8 lines:

```
    # Begin Measure-Text.ps1
        Param ([string]$filename)                                  # Script
with one parameter, a string.
        Write-Output "`nStatistics of $filename `:"                # To return
array
        cat $filename | measure -line -word -character|ft          # To return
array, counts of lines, words, and chars
        Write-Host "You will hear a beep after 2 sec:"             # To console
        sleep -sec 2
        Write-Host 'Beep'; [console]::beep(800.1000);              # To console
    # End Measure-Text.ps1
```

Compare what is written on the console when the output is redirected (the script measures its own length):

```
    .\Measure-Txt.ps1 -filename .\Measure-Txt.ps1 > foo.out
    cat foo.out
```

to when the script is called directly (parameter name `-filename` is optional and omitted):

```
    .\Measure-Txt.ps1 .\Measure-Txt.ps1
```

Variables and functions have a default scope which may be modified. For example, inside a script the following function has only the script as scope. The scope of the variable `$a` is modified to global and hence `$a` is known to the invoking PS session:

```
    function Display-Hello {
      "Hello. World"
      $global:a = 13
    }
    Display-Hello  # --> Hello. World
    $a             # --> 13
```

To contents

26Traversing the Windows Registry

PowerShell is eminently suitable for traversing the Windows Registry. The Registry stores information needed by the programs installed in a Windows environment. It contains the following "hives";

- HKEY_CLASSES_ROOT (HKCR)
- HKEY_CURRENT_USER (HKCU)
- HKEY_LOCAL_MACHINE (HKLM)
- HKEY_USERS (HKU)
- HKEY_CURRENT_CONFIG (HKCC)

Each hive is a tree consisting of keys and subkeys that define traversable paths. Unless a subkey is the end of a path, it contains one or more subkeys that point further down their

paths. A subkey may also contain value entries, which are name-value pairs that offer the desired information to installed programs. The value entries are the very reason for the existence of the Registry. In fact, the tree structure is only a means to help locate the value entries.

The only Registry hives predefined as PSDrives are `HKCU` and `HKLM`. The hives `HKCR, HKU`, and `HKCC` are not directly accessible by `cd`. One must issue `cd registry::hkcr` to access `HKCR:`, etc. The latter change of directory leads to the very long prompt:

```
PS Microsoft.PowerShell.Core\Registry::HKCR>
```

Alternatively, one can define a new PSdrive by a command like:

```
New-PSDrive -PSProvider registry -Root HKEY_CLASSES_ROOT -Name HKCR
```

followed by `cd HKCR:`. The advantage is the much shorter prompt string: `PS HKCR:\>`.

A Registry key that does not contain value entries is called empty. An empty key always contains one or more subkeys. An end node of a path is never empty, it always contains value entries, but by definition no subkeys. Clearly, value entries are not part of a path.

For use in the following examples, set once and for all:

```
set ps 'PSChildName' -opt constant
```

so that from hereon `$ps -eq 'PSChildName'`. In the examples below the current PSdrive is `HKCU\software`. One gets there in a PowerShell session by issuing

```
cd hkcu:\software
```

Recall that `-ea 4` stands for `-ErrorAction Ignore`. This flag suppresses errors about non-accessible keys of which there are many in the Registry. The alias `gp` stands for `Get-ItemProperty`. The alias `gi` stands for `Get-Item`. The functionality of `gi` overlaps to a large extent with `ls` and `gp`.

Now follows a list of examples that may be useful in inspecting/traversing the Registry:

```
    ls -rec -depth 1  -name     # Tabulate names of subkeys (children) and
subsubkeys (grandchildren).
    gp *                        # Lists all value entries (name and value)
in all subkeys plus a few PS variables that define
                                # three generations of the current path.
Skips empty subkeys.
    ls 7-zip | gp               # Value entries (names and values) of the
subkeys of 7-zip (plus a few PS variables).
    gp .                        # Lists value entries in present key. No
output if present key is empty.
    gp * |ft $ps                # $ps='PSChildname' is a PS variable,
returned by gp, that contains the name of subkey (child).
                                # Hence this tabulates names of all non-
empty subkeys.
    gp microsoft                # No output because no value entry present
in subkey 'microsoft' (is empty).
    gp RegisteredApplications  # Value entries (names and values) of
hkcu:\software\RegisteredApplications + PS info
    gi *                        # Same as ls *; lists names of subkeys (also
empty ones) and their value entries;
```

```
                                # no output when present key is endnode.
    gi .                        # Value entries of present key. Almost same
info as 'gp .'. but somewhat different format.
    gi .\Microsoft\Notepad      # Value entries of subkey
    gi .\microsoft\Notepad |fl *   # A subkey is an object. list its
property names and values. One member is array 'Property'
    gi .\microsoft\Notepad |select -exp property # The content of array
'Property' in expanded form.
                                                  # the array elements
contain the names of value entries of the present key.
```

Note on ls * −rec −depth 1

Consider the following two commands issued from `HKCU:\software`:

```
    ls * -rec -depth 1 -ea 4 | measure -line   # gives 35329 lines
    ls   -rec -depth 1 -ea 4 | measure -line   # gives   804 lines
```

while both commands—issued from `c:\`—give the very same number of lines (542). It is difficult to see this dependence on context as anything but a bug. It is, therefore, advisable to never use `ls *` together with the flags `-rec -depth`.

End note

The following command lists names of subkeys and subsubkeys of the present key together with an array containing the names of their value entries:

```
    ls -rec -depth 1 |select name, property
```

Here `property` contains the names of all value entries, but only the first few elements are listed. To expand this array, together with the names of the subkeys separated by an empty line, use the following:

```
    ls -rec -depth 1 |select name, property |% {$_.name; $_.property; "" }
```

Explanation: `ls` returns an array of subkey objects that all have the properties `name` and `property`. The cmdlet `select` picks from each subkey object these two properties and adds them to an object that is referred to by `$_` in the next stage of the pipeline. These objects are collected in an array that is passed to `% = Foreach-Object`. Then `%` loops over the array elements. The script block in the body of the loop simply issues the two member names as commands. Issuing of a variable name gives the writing of the content of the variable. If the content is an array, the array is written element for element, every element on a new line.

To contents

27Registry lookup

If a value entry or a subkey must be located, it is necessary to descend down hive trees. The cmdlet `Set-Location` (alias `cd`) enables this. The `-recurse` parameter of `ls` does not imply any `cd` in a script block. An explicit `cd` is necessary if some processing must be performed lower down the path. For example, recalling that `ls -name -rec -de 1` returns an array containing names of subkeys and subsubkeys, we see that the following statement gives the names of all subkeys and subsubkeys by execution of `pwd` (which returns the name of the present key). including those without value entries (the empty ones):

```
    ls -name -rec -depth 1 |% {$p=pwd; cd $_ -ea 4; pwd; cd $p; rv p}
```

Here `-ea 4` suppresses the listing of an error when a `cd` to a non-accessible subkey is attempted. The newly created variable `$p` is removed to avoid possible later side effects. Compare this statement to the following command that lists names (contained in `PSPath`) of *only non-empty* subkeys and subsubkeys, but does not require a `cd`:

```
ls -name -rec -depth 1 |gp |select pspath
```

Once one has descended to a certain key, the names of its value entries (if any) are obtained by:

```
gi . |select -exp property  # This returns line by line the names of the value entries.
```

The command `gpv -name entry_name` returns the value coupled to `entry_name`. For instance, the subkey `7-zip` of `HKCU\software` contains the value entry pair `(lang: nl)`, i.e., it has entry name `lang` and entry value `nl`. The commands issued from `HKCU\software`:

```
$p=pwd; cd 7-zip; gpv -name lang; cd $p; rv p;
```

effectively leave us in `HKCU\software` and outputs the entry value `nl` named `lang`.

To list the value entries (if present) of the present key, use

```
gi . |select -exp property |% {$v=gpv -name $_; write-host $_":", $v; rv v;}
```

The next command lists names (`$p`) of non-empty subkeys, subsubkeys, and subsubsubkeys, name of value entry (`$name`) and corresponding value (`$v`). It does not list the value entries of the current key. Note the nesting of `|%` and the line continuation:

```
ls -rec -name -de 2 |% {$cd=pwd; $p=$_; cd $p -ea 4; gi . |select -exp property|`
% {$name=$_; $v=gpv -name $name; write-host $p": ", $name" = "$v; rv name, v }; cd $cd; rv cd, p}
```

The final command finds entry values containing a given string in keys below the current key. It is important to note that the search time increases exponentially with the value of the parameter `-depth`:

```
$string = "aul"
ls -rec -name -depth 2 |% {$cd=pwd; $p=$_; cd $p -ea 4; gi . |select -exp property|`
% {$name=$_; $v=gpv -name $name; if ($v -match $string) {write-host $p": ", $name" = "$v;}; rv name, v }; cd $cd; rv cd, p}
rv string
```