Windows PowerShell Cheatsheet.

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.*

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. Unrelated to PS

A few lesser known windows features and tricks, not directly related to PowerShell:

1. In the Windows (File) Explorer address bar enter `cmd`, `powershell.exe`, or `powershell_ise` and a corresponding window opens at the current directory (aka folder). This is most likely the shortest route to opening a CMD, PS, or PS_ISE session from a given directory.
2. After hitting the start or search button in the windows taskbar type any of: `Control Panel` (in Dutch: `Configuratiescherm`), `Regedit`, `PowerShell`, or `cmd`. This opens a window with the name of the chosen program (`Control Panel, Regedit`, etc). Click on it once and the program chosen opens.
3. Adaptation of user environment variables: `Control Panel/User Accounts/User Accounts/`. Then left panel: `change my environment variables`. (Choose in `Control Panel: view by category`).
4. Adaptation of system environment variables. Choose in `Control Panel: view by category`. Then `Control Panel/System and Security/System/Advanced System Settings`. Button: `Environment Variables`. Set in panel `system variable`. Change is persistent, a change (see below) within PowerShell is volatile.
5. In a CMD session: `setx` sets persistent user environment variable. To unset: go to `HKCU/environment` (in the Registry) or to `Control Panel/User Accounts`.

To contents

2. About ISE

ISE stands for Integrated Scripting Environment. ISE—a standard part of Windows 10—is an editing and execution environment for PowerShell.

An important advantage of the use of ISE over pure PowerShell, is that ISE offers pop-up help: when the user is typing a command line, ISE often pops up a relevant list of methods and properties. By double clicking on an entry in the pop-up list, the entry is appended to the command line.

A few noticeable points:

1. Upon the start of ISE, the script `Microsoft.PowerShellISE_profile.ps1` is executed.
2. To protect ordinary users against malignant PS scripts, the execution of PowerShell scripts (including the ISE profile) must be

allowed explicitly. This is done once and for all by issuing the command
```
Set-ExecutionPolicy RemoteSigned -Scope CurrentUser
```
3. 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).
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. Set width and height of the ISE output pane by:
```
$Host.UI.RawUI.BufferSize = New-Object $Host.UI.RawUI.BufferSize.GetType().Fullname (150, 40)
```
   (The pure PS console does not allow shrinking of width).
6. Set ISE console fontsize by:
```
$Host.PrivateData.FontSize = 13
```
   (pure PS has a different `$Host.PrivateData` object).
7. Start PowerShell as administrator:
```
Start-Process PowerShell -Verb runAs
```
8. An important ISE tool for PowerShell learners is CTRL-J. This pops up a selection of code snippets. These are templates that give illustrative examples of PS syntax and can be used in scripts as a start points for further editing.

To contents

3. Cmdlets

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-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).
```

To contents

4. PSdrives

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` is a hive in the Registry.

Standard 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

5. Pipelines

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.

The output of `Get-ChildItem flop.txt` is an object and by the pipe

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

a full list is obtained of the members (in object-oriented languages known as *methods* or *properties*) of the object associated with the file `flop.txt`. In this list we see that one of the members 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, by adding the flag `-name`

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

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

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

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

However, if the piped object `$_` does not have the member (property) `name`, an error occurs:

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

When more than one object is outputted by a cmdlet in the pipeline, these objects are first stored in a temporary buffer. After the buffer is filled, the cmdlet in the next stage performs its task by looping over the buffer and reading the objects one by one. When this stage is not the last of the pipeline, the cmdlet puts its outputted objects in yet another temporary buffer that serves as the input for the next stage. An object that drops out at the end of the pipeline is usually written to screen by a screen writing method of the object. The concept of intermediate buffering is very important for the understanding of PowerShell 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 stores them in a 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 *
```

The result on the screen is a list of the names of all files in the current folder. The list is in alphabetic order.

The use (or missing) of parameters of cmdlets that receive input in pipelines can generate unexpected errors. Example:

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

moves all `.txt` files to folder `subdirectory`. Analogously, one could assume that the following pipe would move the files one folder up:

```
     Get-ChildItem *.txt | Move-Item ..\   # Error!
```

This gives the error message that `Move-Item` lacks input:

```
   Move-Item : The input object cannot be bound to any parameters for the command either because the command does not take
   pipeline input or the input and its properties do not match any of the parameters that take pipeline input.
```

The parameter `-destination` remedies this and

```
    Get-ChildItem *.txt | Move-Item -destination ..\
```

moves without complaint all `.txt` files one level up.

To contents

6. Useful aliases

Many cmdlets have one or more aliases. Often an alias is DOS- or Unix-like.

```
    ac = Add-Content                 # Example: ac -value 'The End' -path 'flop.txt'
                                     # (appends value to file)
    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 = Remove-Item        # Remove files, registry keys, etc.
```

```
                                 #
dir = gci = ls = 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
                                 #
fl = Format-List                 # Example: ls env:Path |fl
                                 # (gives 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 (name without $ prefix, while
                                 # note that variable names must begin with $)
select = Select-Object           # Select specified properties of piped object
                                 # Example: ps |select Processname | select -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 pipeline.
                                 # Example: ls -recurse |? name -like '*Pict*'
$env:userprofile = ~             # Example: cd ~ (change folder to home folder of user).
```

To contents

7. Example 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:

```
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 `$a`. (Recall here that variable names are text strings that begin with a dollar sign). Note that the cmdlet `Where-Object` (alias: ?) can recognize names of object properties without use of a script block. That is, `{$_.length -lt 10000}` is equivalent to `length -lt 10000`. Thus,

```
$a = ls -r *.dll |? length -lt 10000  # Store in $a 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
                                                  # formatted name, length, and directory
```

Finally, in one statement:

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

To contents

8. More 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 regexp:

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

(Note that 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 but wrapped.

Find all directories 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

9. Examples Where-Object

The cmdlet `Where-Object`, which only appears in pipelines, has alias "?".

As an example, list the basenames ending with the letter `r` in the current directory. Use the operator `-match` with a regexp and recall that the symbol `$` indicates the end of a string:

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

This lists the basenames ending on 'r' plus additional information (mode, write time, length, full file name). If only the basename is to be listed, use:

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

Example of the comparison operators: `-in, -notmatch, -and, -notlike`. The first statement below limits the list of 'svchost' and 'firefox' processes to the first 10 in total. The next statement uses a script block, which is the code snippet between curly brackets:

```
    ps |? ProcessName -in  "svchost", "firefox" | select -f 10 |ft processname,  PagedMemorySize
    ls |? {$_.name -notmatch 'e$' -and $_.name -notlike 'c*'}     # Names not ending on "e" (regexp) or beginning  with "c"
```

To contents

10. Examples ForEach-Object

Remember that `%` is an alias for `ForEach-Object`, as is `foreach`. Example:

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

This gives:

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

`Get-Alias` sends 158 objects (aliases) through the pipeline. All objects have a property `name`, which is matched against a regexp. The regexp checks if the first letter is 's' or 'S'. If true, the object attribute `name`, the string "stands for:", and the object property `definition` are written. The `Write-Host` cmdlet, which allows a certain freedom in the format of its output, writes to a host—which in an interactive session is the screen. A boolean expression must be surrounded by round brackets, the body of the Foreach-Object must be enclosed by curly brackets (the outer ones), and the body of the true branch is within curly brackets too.

Note that a list of aliases starting with 's' or 'S' can be obtained in a prescribed format from the shorter `Where-Object (?)` statements:

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

And still shorter:

```
    gal s*
```

More than one statement may be in the body of the `ForEach-Object` loop; statements are separated by semicolons. 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`. Remember that `cat` returns an array object that has the method `count`.

As yet another example, notice first that

```
    gal cat |? displayname
```

displays the string

```
    Alias           cat -> Get-Content
```

The verb "Get" can be captured from this string by the comparison operator `-match` with the regexp (note the parentheses indicating a capture):

```
    $reg = '-> (\w*)-'
```

and the captured result appears in `$matches[1]`. Here `\w*` indicates an arbitrary number of word characters. The alias `gal (Get-Alias)`, without arguments, fills a buffer containing all aliases. Each object in the buffer has the property "displayname". Hence to get a sorted list of the unique verbs of all aliases, issue:

```
    gal|? displayname -match '-> (\w*)-' |% {$matches[1]} |sort -unique
```

Three aliases are missed here (namely `ise, man, md`) because their definition lacks the hyphen: `-`.

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
```

To contents

11. Select-String

We met the the cmdlet `Select-String` (alias `sls`) above. Because it is an extremely useful tool—and not an easy one as it depends heavily on regular expressions—we give three more examples of its use.

Consider as a first example the statements

```
$address = 'https://131.174.138.39/~pwormer/teachmat/PS_cheat_sheet.html'
$parsed  = $address |sls -patt 'https://([0-9.]*)/(.*)$'
```

The second statement breaks out the IP address and the path from `$address` and assigns them to submembers of the object `$parsed`. This object has several members, one of them `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    : https://131.174.138.39/~pwormer/teachmat/PS_cheat_sheet.html
```

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

```
$parsed.matches.groups[0].value : https://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
```

As another example, we remark that the present file, called `PS_cheat_sheet.html`, contains strings that are enclosed in <b> and </b>, i.e., the strings are displayed in bold face. We will now construct a PS pipeline that lists these strings. (The pipeline does not work correctly when bold text extends over different lines. Moreover, the search will be non-greedy, which means that only the first bold string in each line is returned). First we introduce a regular expression (regexp) that captures bold HTML text:

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

This regexp matches the first arbitrary string in a line that is enclosed within `<b>` and `</b>`. It captures the string matching the regexp inside the parentheses. The latter regexp matches zero or more (`*`) arbitrary characters (`.`) in a non-greedy search (`?`). The regexp `$reg` is used in:

```
sls PS_cheat_sheet.html -patt $reg |% {$_.matches.groups[1].value}
```

which writes all the bold strings in the present notes.

As a final example we consider the file `foo.txt` with contents:

```
In columns 20...25 (six columns) are positive, negative, and unsigned integers.
   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 statements list 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 regexp skips arbitrary characters in the first 19 columns and captures digits and plus/minus signs in columns 20-25. The sum is accumulated in `$s` where PowerShell converts the captured strings to integers.

To contents

12. Create 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 we create a symbolic link on the desktop, named `manual`. (In Dutch Desktop is *Bureablad*). 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\paul\Bureaublad\manual -Target .\manual.txt
```

This can be shortened to:

```
ni -ty sym -p C:\Users\paul\Bureaublad\manual -v manual.txt
```

The parameter `Target` has the alias `Value`, `ItemType` has the alias `Type`, 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 $_}
```

To contents

13. Rename 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` 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 `-replace` operator. Its syntax is:

```
   string -replace regexp, new_name
```

In `string` every substring that matches the regular expression `regexp` is replaced by `new_name`. For example,

```
     'report.txt' -replace '\.txt$', '.doc'  # -> 'report.doc'
```

The regular expression `'\.txt$'` is anchored at the end of the string by "`$`" and the dot is escaped so that its meaning is not the regexp arbitrary character but the ordinary punctuation mark. Thus,

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

replaces in the current directory all file extensions `.txt` by `.doc`. The value of the script block (the part between curly brackets) is a string: the new file name.

In the previous example the PS statement is considerably more difficult to memorize than the equivalent CMD statement. Because that happens more often, 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 *.doc *.pdf
```

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

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

This statement is not easy to memorize either.

Note. The text with hyperlink `-replace` operator states explicitly that the first argument of `-replace` must be a regular expression. In the fourth example of

```
    get-help rename-item -examples
```

the first argument of `-replace` is `'.txt'`. At first sight this looks like an ordinary string, but if we remember that the dot stands for an arbitrary character, we do recognize it as as a regexp.

To contents

14. Datatypes

Type cast operators are among others: `[int], [long], [string], [char], [bool], [byte], [double], [decimal], [single], [array], [xml], [hashtable], [PSCustomObject].`

Once a variable is declared explicitly, an implicit type change is forbidden. Examples:

```
    [string]$s = 'Peanuts'    # Explicit declaration
    $s         = 4            # Assign new string '4'
    $s * 3                    # Gives 444 (triplicates the string)
    [int]$s    = 4            # Explicit change 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 is OK on RHS (hex-char 0x07 sounds "Bell", warning sound)
```

Another type casting example:

```
    'a', 'b', 'c', 'd' > letters.txt       # Each letter on a line in file letters.txt. Output in UTF-16 (!)
    [string]$letter = cat letters.txt      # Get-Content, assign to string
    $letter                                # --> a b c d   (a single string)
    $letter = [string](cat letters.txt)    # The same. Brackets are necessary to cast the output of the cmdlet
```

Use `set` to assign a constant:

```
    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
    $pie                      # -> 3.14159265358979
```

Arrays

```
    $a = @('a', 'b')          # Array    (round brackets, colons as separators)
    $a += 'c'                 # Push 'c'
    $a += 'd'                 # Push 'd'
    $a                        #  --> a\n b\n c\n d  (vertical stack)
    $a.GetType()              #  --> True      True     Object[]      System.Array

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

Associative arrays (aka dictionaries or hash tables)

```
    $assoc = @{one=1 ; two=2} # Associative array (curly brackets, semicolons as separators)
    $assoc.one                # -> 1
    $assoc['two']             # -> 2
    $assoc['three'] = 3       # Adds a member
    $assoc.four = 4           # Adds a member
    $assoc.GetType()          # --> True      True     Hashtable     System.Object
    $assoc                    #
```

The last statement (`$assoc`) gives:

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

To contents

15. Strings

Strings are as in 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
```

Backtick escapes under 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`. The file `out.txt` will be in UTF-16.
3. The cmdlet `add-content` (alias `ac`) writes to file by default in ANSI (Windows-1252), and allows specification of other encodings.

Here-strings
`@' ... '@` (no expansion) and `@" ... "@` (with expansion). Note that 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
```

To contents

16. The formatting of strings

The PS formatting of strings is derived from the composite formatting of `C#`. The format operator is `-f`. Schematically it is used like:

```
"String containing format-items"  -f elements to be formatted separated by commas
```

Examples:

```
"{0, 0:f4} is rounded to 4 decimals" -f 12.34567       # -> 12.3457 is rounded to 4 decimals
"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
```

On the left-hand-side of the operator `-f` there is a string containing format-items:

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

[Square brackets surround optional values]. The index is 0-based and refers to the position in the array on the right of `-f`. The alignment gives the total number of spaces reserved for the output, with the output string being 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.

A composite-formatted string can be written by `Write-Host`:

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

Note the order: second argument first, then the first, and finally the third. Also look at the spacings 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 they may be appended by an integer giving the number of decimals:

:c	Currency (output depends on the 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 point and thousands separators (choice of dots or commas depend on the region set in Windows Control Panel)
:p	Percentage
:x	Hexadecimal format (only integers)

To get the sizes of all PS-scripts in the current directory (`gp = Get-ItemProperty`):

```
gp *.ps1 |% {$_.length}
```

A script block allows computation (`1kb` is a literal constant of value 1024):

```
gp *.ps1 |% {$_.length/1kb}     # Lengths in kilobyte
```

List basename and length in a pretty format (round brackets are needed to give priority to the division):

```
gp *.ps1 |% { "{0, 31} {1, 6:f1}" -f $_.basename, ($_.length/1kb)}
```

As usual, there is a simple alternative to get (almost) the same info (lengths in bytes):

```
ls *.ps1 |ft basename, length
```

Strings have methods, if we enter

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

we see lots of string methods 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.

To contents

17. Comparison

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   (regexp: all letters)
'peanutbutter' -match 'r$'         # true   (regexp: 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'
```

A website may contain the microsoft installation app `UsefulApp.msi` and its SHA256 hash in the text file `UsefulApp.msi.digest` (64 hex digits). Download both and issue:

```
(Get-FileHash .\UsefulApp.msi).Hash -eq (cat .\UsefulApp.msi.digest)[0]
```

If this returns `True`, you can trust that the downloads haven't been changed on their way down to your computer. The cmdlet `Get-FileHash` returns an object with 3 members, one of them called `Hash`. Because `cat` returns an array, you need to pick out its first element. Note parenthetically that when the publisher of `UsefulApp.msi` uses MicroSoft Authenticode, the cdmdlet `Get-AuthenticodeSignature` may be worth looking at.

To contents

18. Switch

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

19. Builtin classes

PowerShell has built-in 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 contents

20. Random numbers

```
Get-Random -min 0.0 -max 1.0  # Random nr between 0.0 and 1.0, see help Get-Random for bounds
```

Alternatively, use the 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
```

Fill the array by a method of instance `$rand`:

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

To contents

21. Errors

Almost all cmdlets recognize the flag `-ErrorAction`, abbreviated: `-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 *.jpg  # all .jpg files in present and all subdirectories
```

As in many languages errors may be trapped. Enter `Get-Help about_trap` to see how. The very same info as web page 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

22. Functions

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" is any of the existing verbs.
To get the unique verbs of all (including user) functions sorted in alphabetical order:

```
gcm -commandtype function |select  verb -unique |sort verb|ft
```

Example of a user function:

```
function Write-ToScreen($path, $name) {
     Write-Host $path, $name
}
```

Alternatively, the parameters can be defined by the `param` statement:

```
function Write-ToScreen {
     param($path, $name)
     Write-Host $path, $name
}
```

Both forms can be called as (parameters can be abbreviated to unique strings):

```
Write-ToScreen -path roaming -name debug.txt    # --> roaming \n debug.txt
```

Or more briefly:

```
Write-ToScreen roaming debug.txt        # Quotes are not required
```

Or more classically:

```
Write-ToScreen('roaming', 'debug.txt')  # Quotes are required
```

With regard to return values: (most) console output generated in the body of the function is collected into an array which is returned. After completion of the function call, the return array may be written to the console (the default), or it may be assigned to a variable, or redirected to a file, or piped to a cmdlet. Indeed, the following three kinds of console output are collected in a return array:

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

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 (do not generate return values):

```
function list-yes{
   Write-Host "first"
   Out-Host -i "second"
}
$a = list-yes    # --> first \nsecond  (vertically stacked) to console
$a               # No output
```

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 = 'A';  $b = 'B'; $c = 'C'; $d = 'D'; $e = '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}
$a = list-txt  # No console output, output of ls returned.
$a             # Info of .txt files to console
list-txt  | ft name  # Only file names to console

function list-txt{ls *.txt|write-host}
$a = list-txt  # Long file names 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 John     # -> John
```

Positional parameters are passed in the automatic array `$args`:

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

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

```
Function Test-PipedValue {
    Process {
        Write-Host "name:  " $_.name,
                   "color: " $_.color
    }
}
```

Example, create hash table and pipe into function:

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

The process command loops over elements of the piped object:

```
function Test-Loop{
    process {
       Write $_    # alias of write-output, appends EOL char
    }
}
1, 2, 3, 4 | Test-Loop # --> 1\n 2\n 3\n \4
```

See `help about_functions` for more info.

To contents

23. Scripts

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
$a
```

To contents

24. Traversing 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, we 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

25. Registry 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
```

To contents